

# In Your Hands : The Character [ of Watch\_Dogs ]

David Therriault

Technical Team Lead Programmer  
Watch\_Dogs, [Ubisoft Montreal](#)

You can find the  
presentation with some  
notes at page 120 in the PDF



UBISOFT

15





Satisfaction Thrill  
Comfort Action  
Joy Precision Ease  
Emotion Excitement  
Power Trip  
Happiness Pleasure  
Achievement Control  
Fun  
Experience





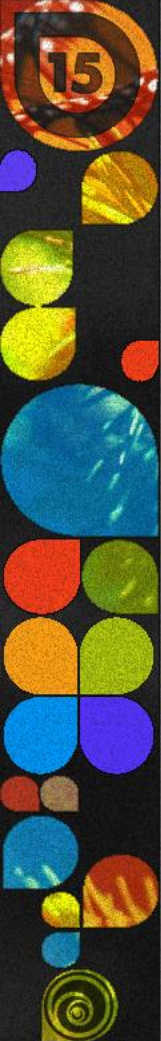
# Immersion Experience







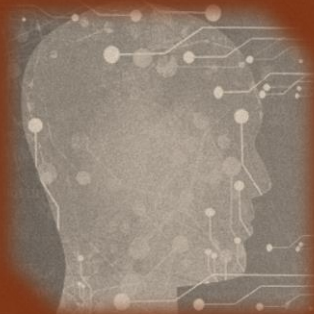




Narrative



Immersion



Game Feel



## Narrative



# Immersion



## Game Feel



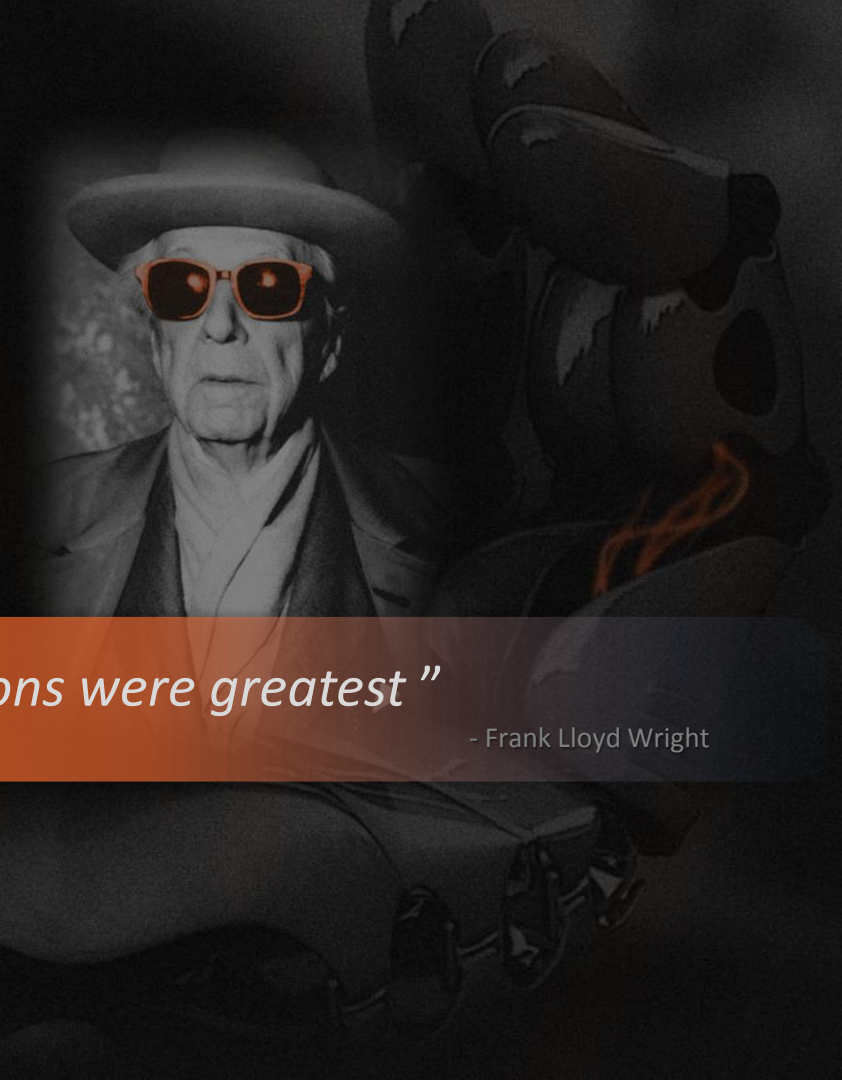
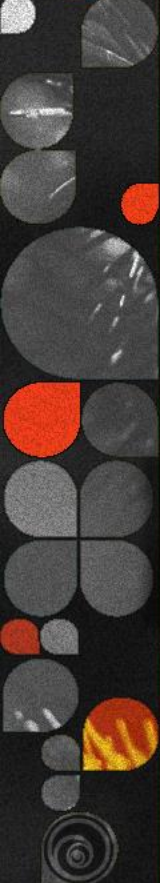
Usability

Grounding

Spatial Awareness

Body Awareness





*“ Man built most nobly when limitations were greatest ”*

- Frank Lloyd Wright







# Density and credibility of **population**







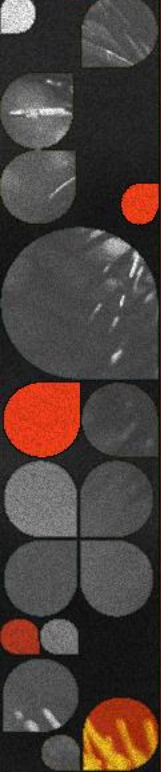
Population being **local** or online





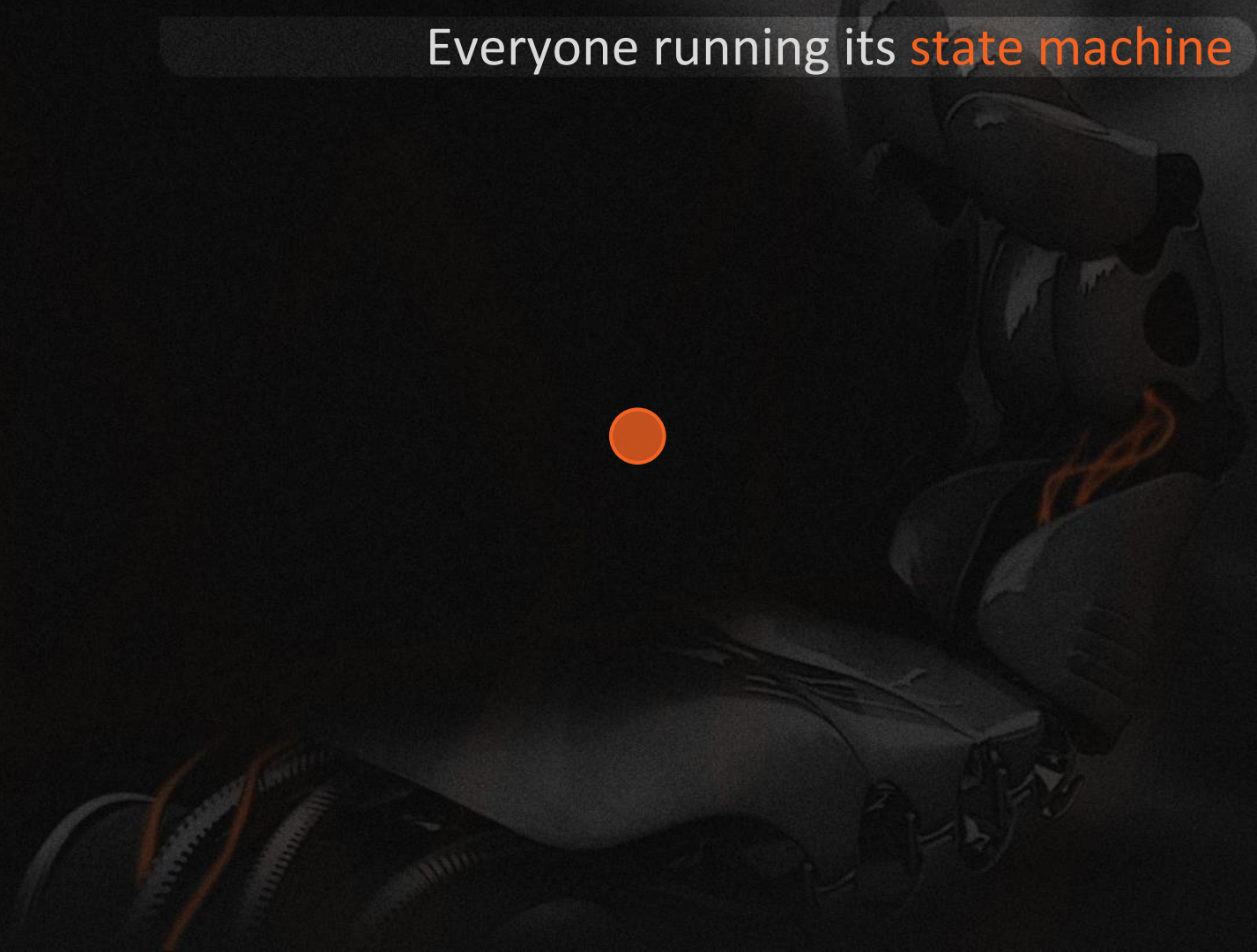


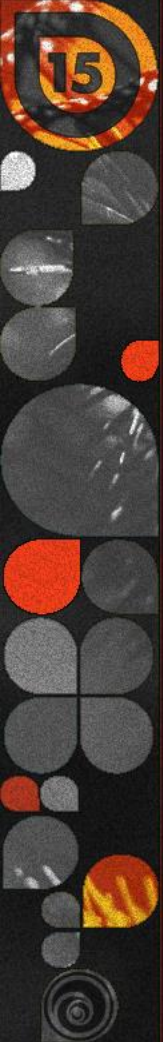
Architecture



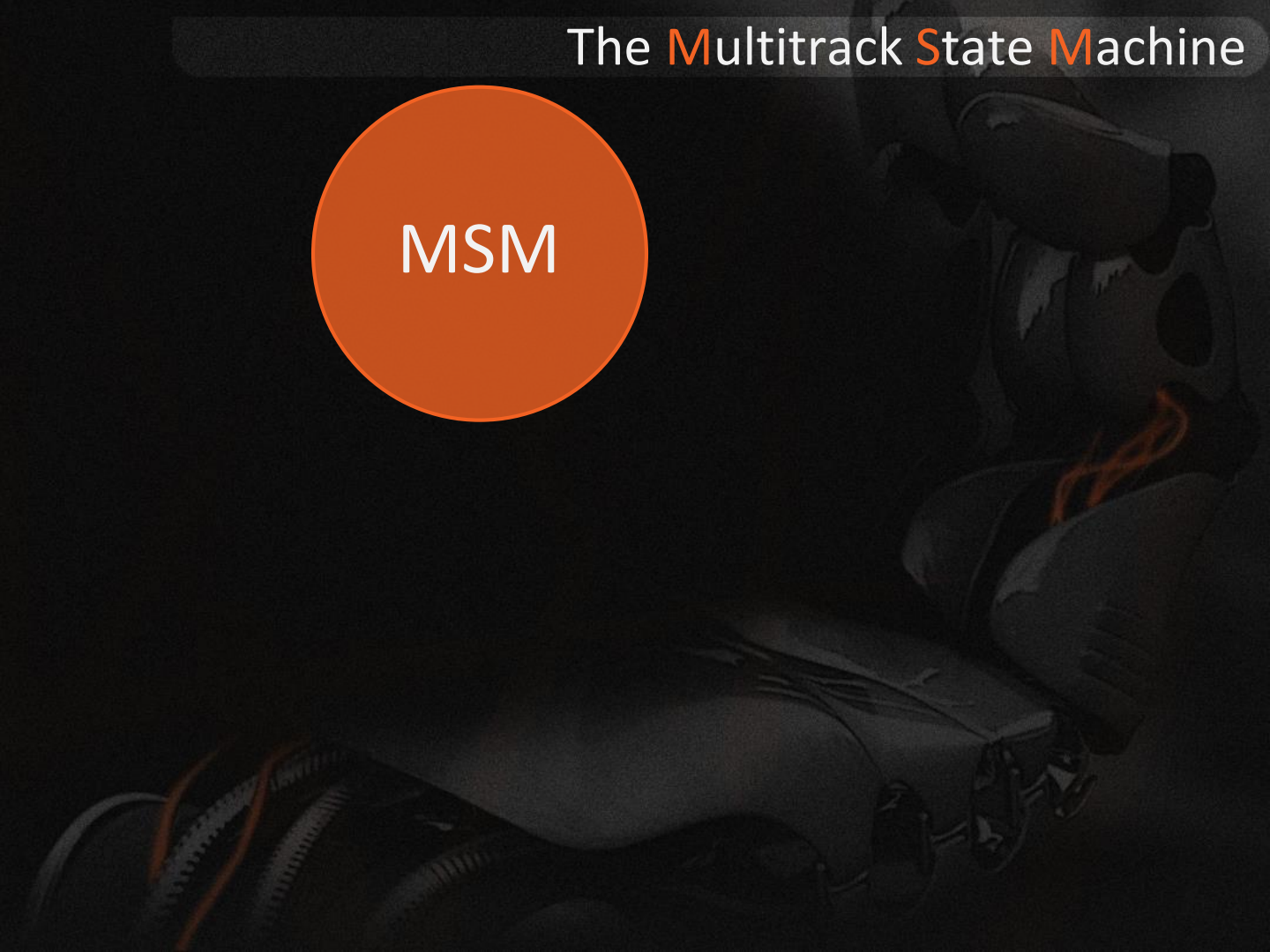
UBISOFT

Everyone running its **state machine**





MSM







Controllers



MSM





## Controllers

- Game Pad
- Behavior Tree
- Path Follower
- Vision System

MSM







Architecture

# The MSM Ecosystem

Controllers

MSM

Sensors



UBISOFT



Controllers

MSM

Sensors

- Frontal Collisions
- Cover Planes
- Jump Annotations
- Stimulus Emissions





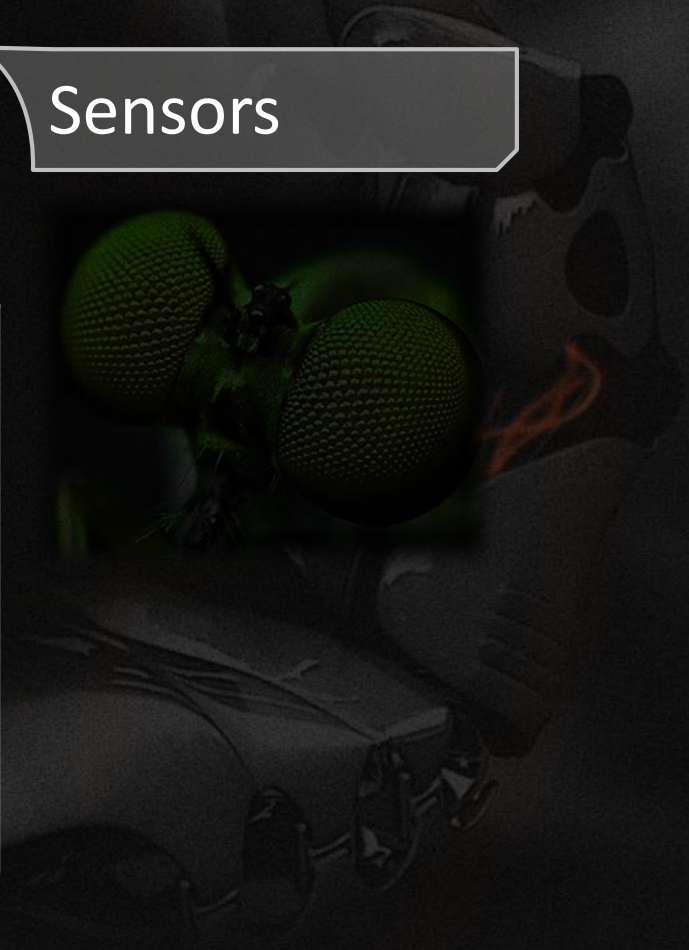


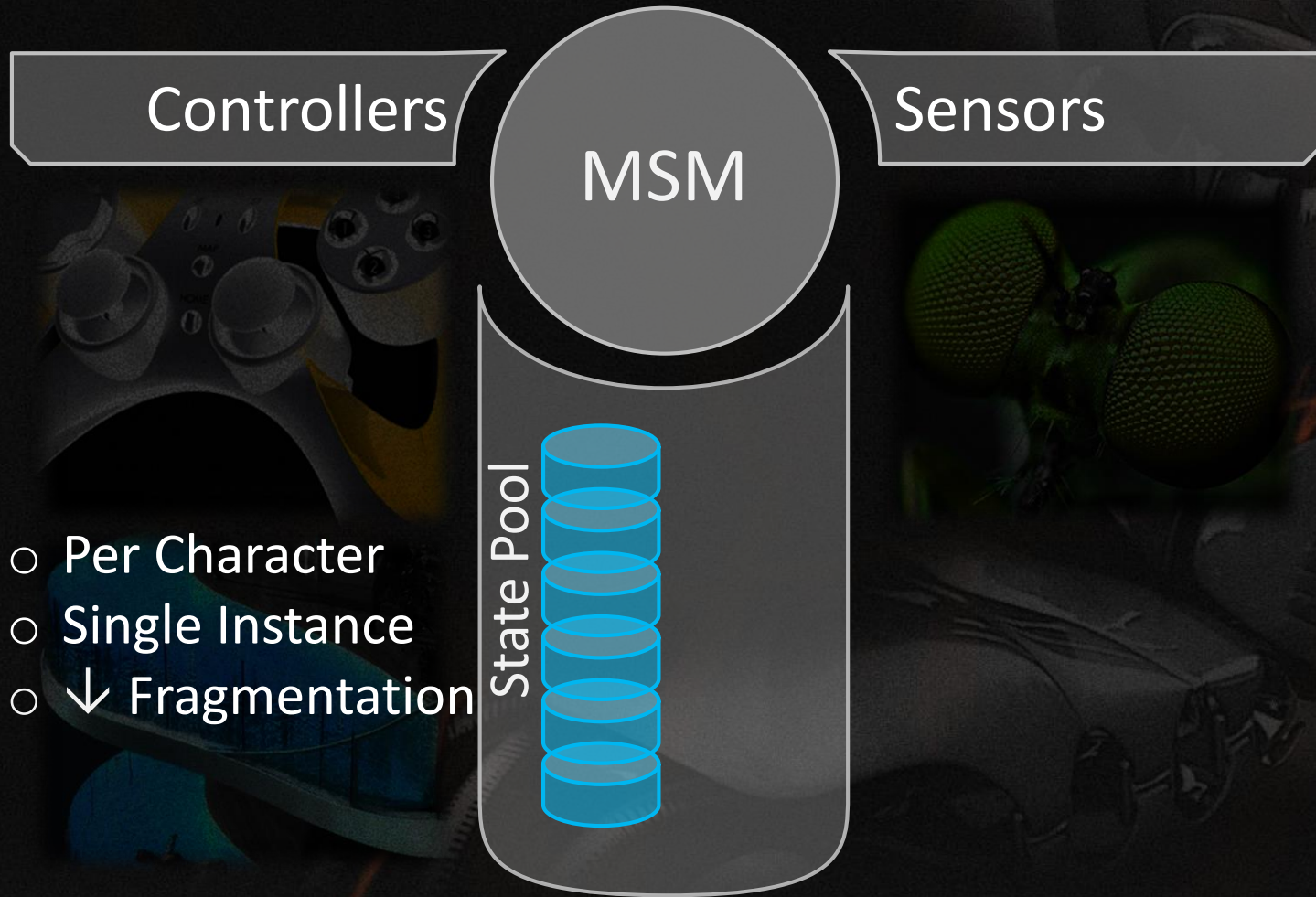
Controllers

MSM

Sensors

State Pool







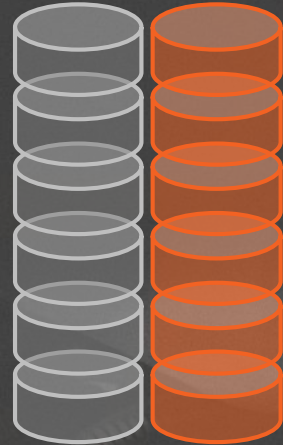


Controllers

MSM

Sensors

State Pool



Data Container

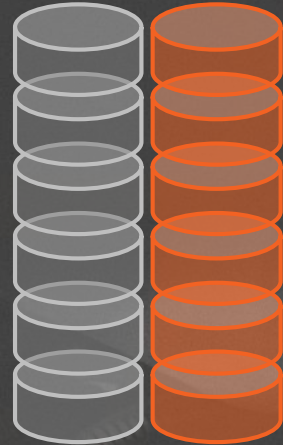


Controllers

MSM

Sensors

State Pool

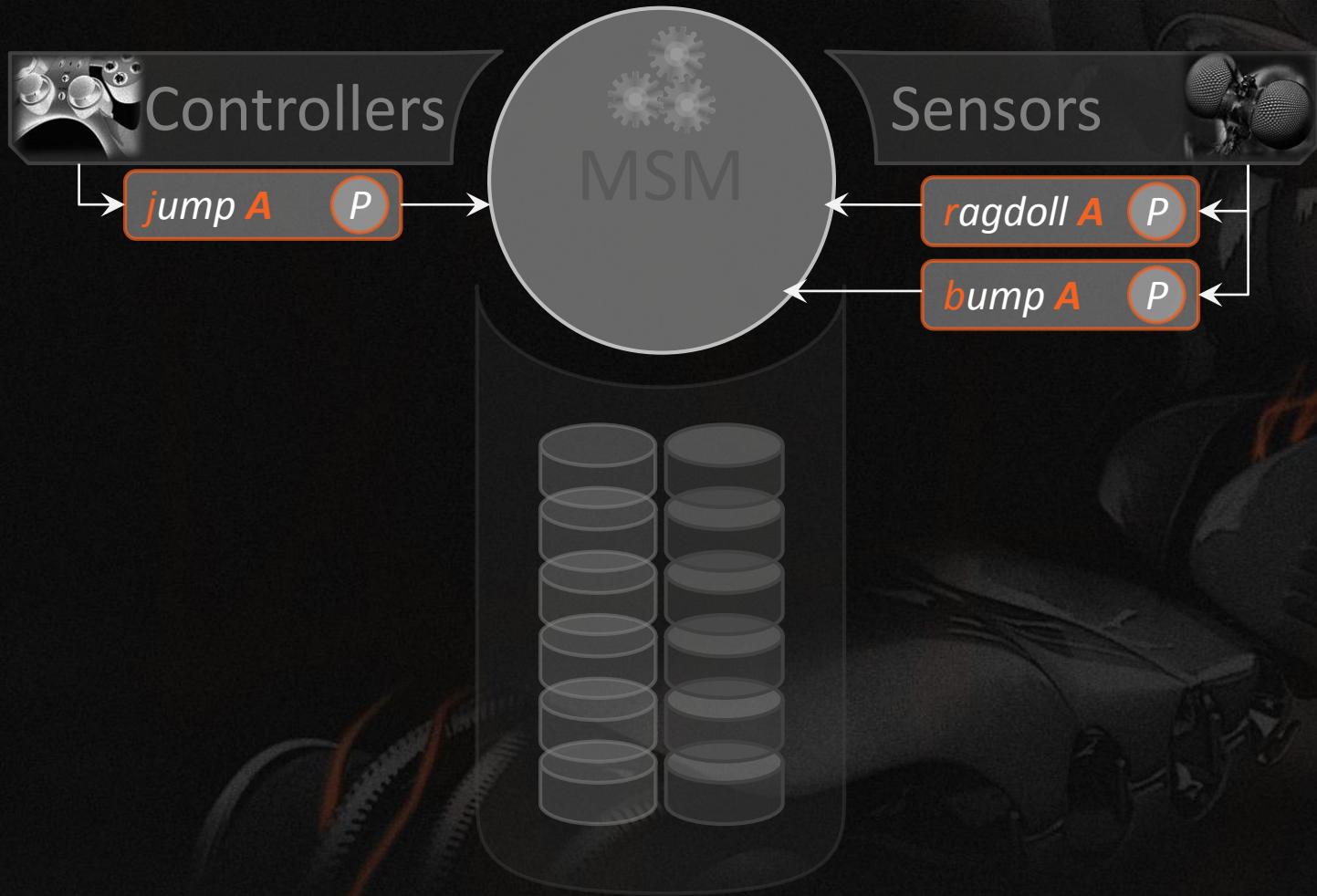
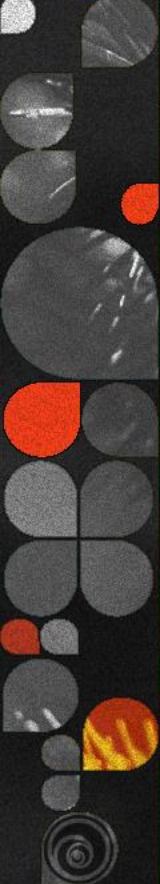


Data Container

- Per System
- Global I/Os
- Partially Replicated

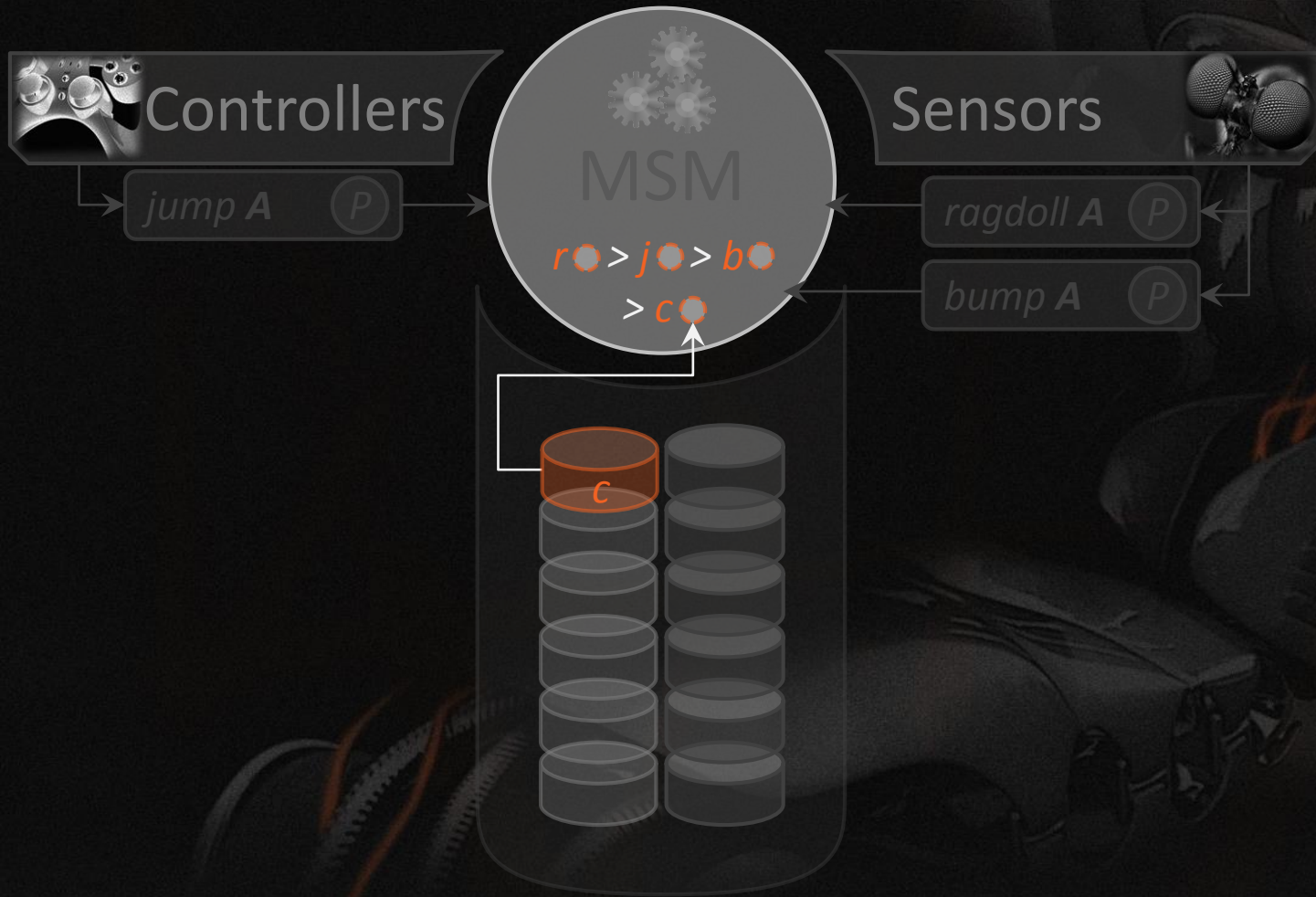




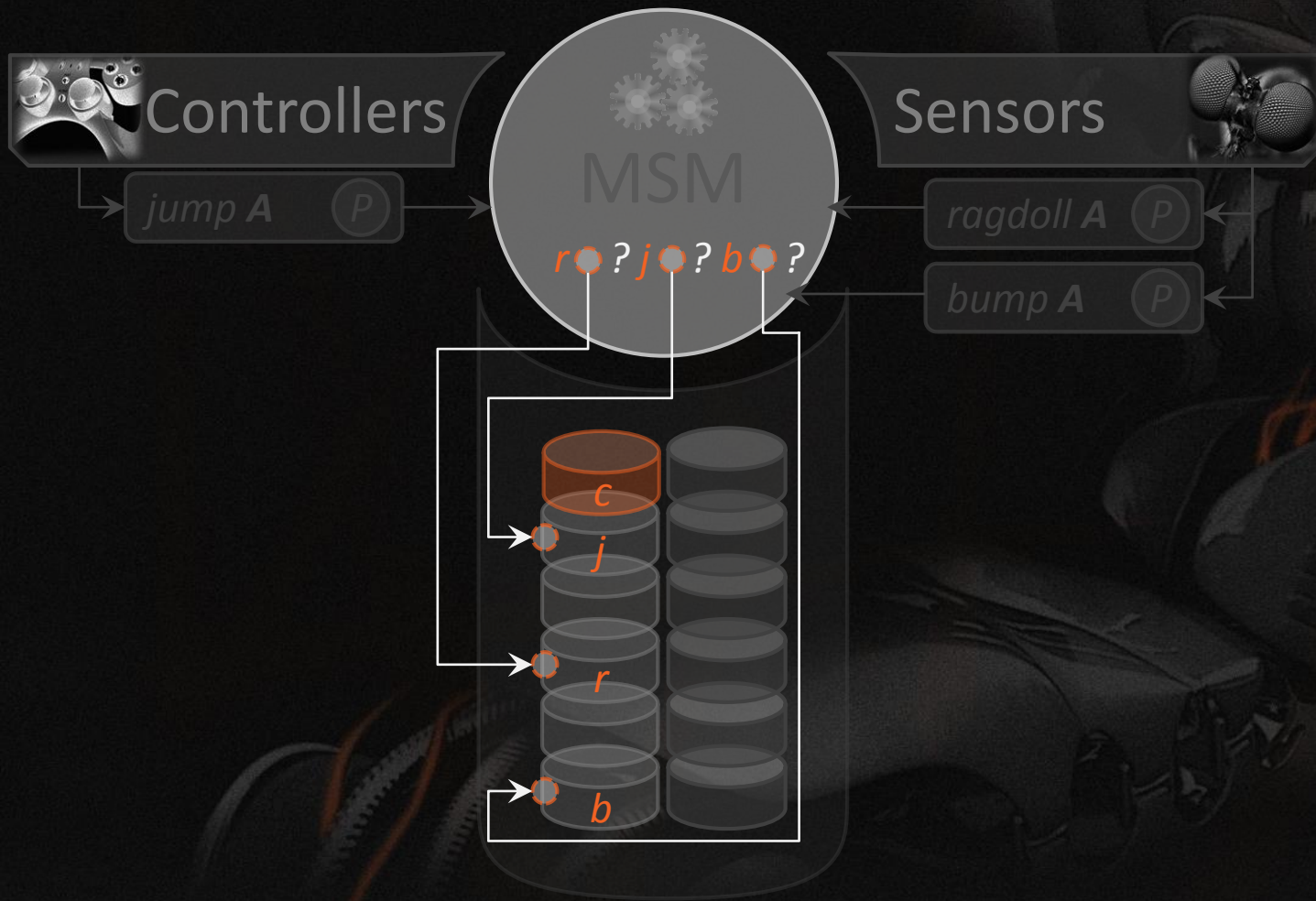


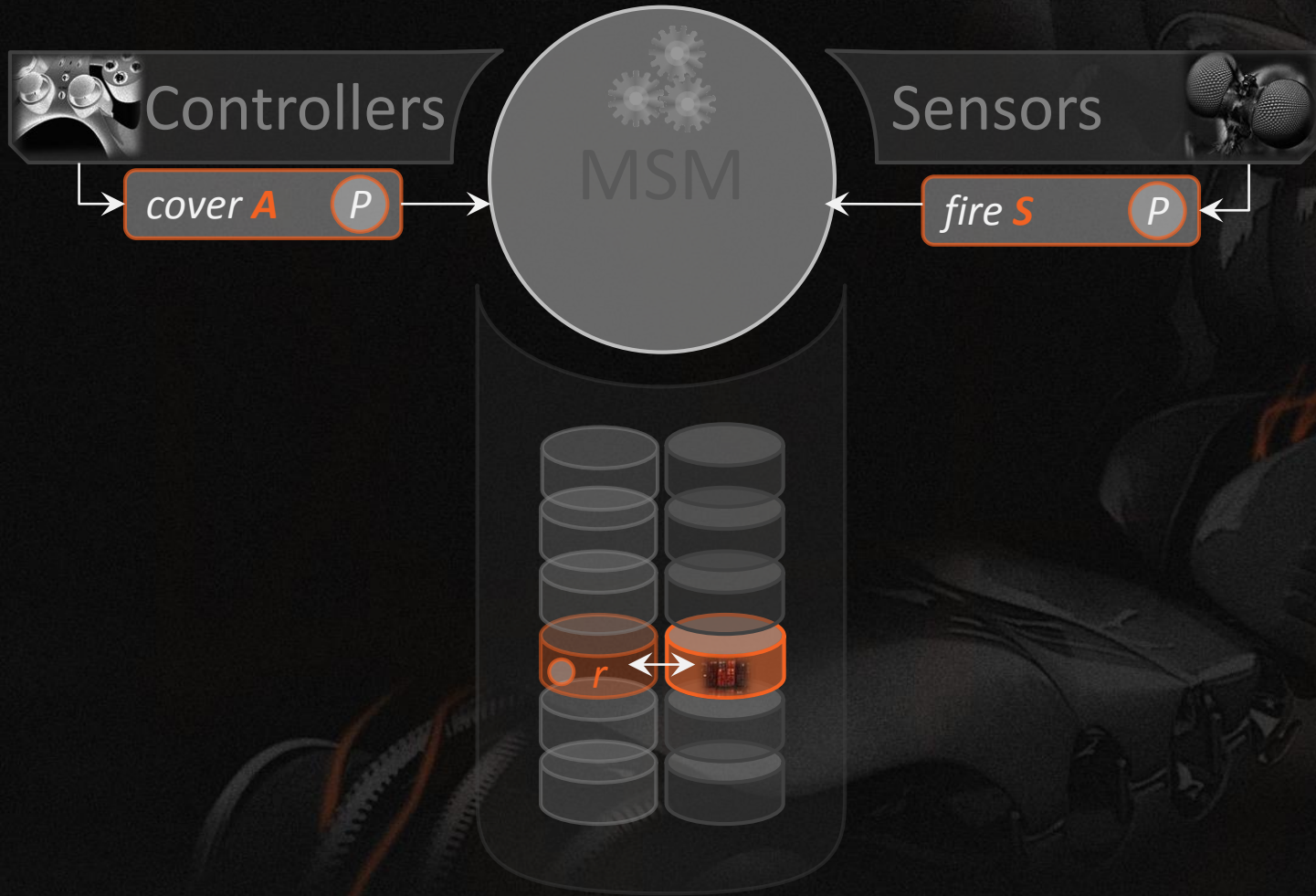
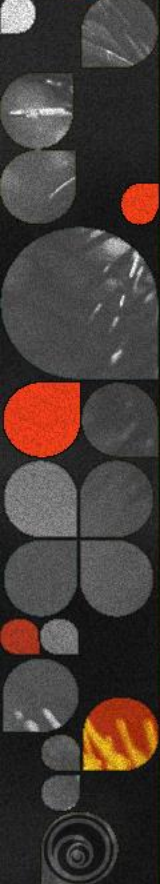


# Stacking & Sorting Requests





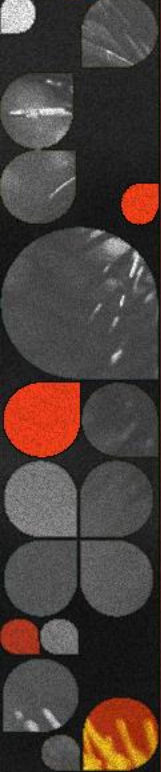






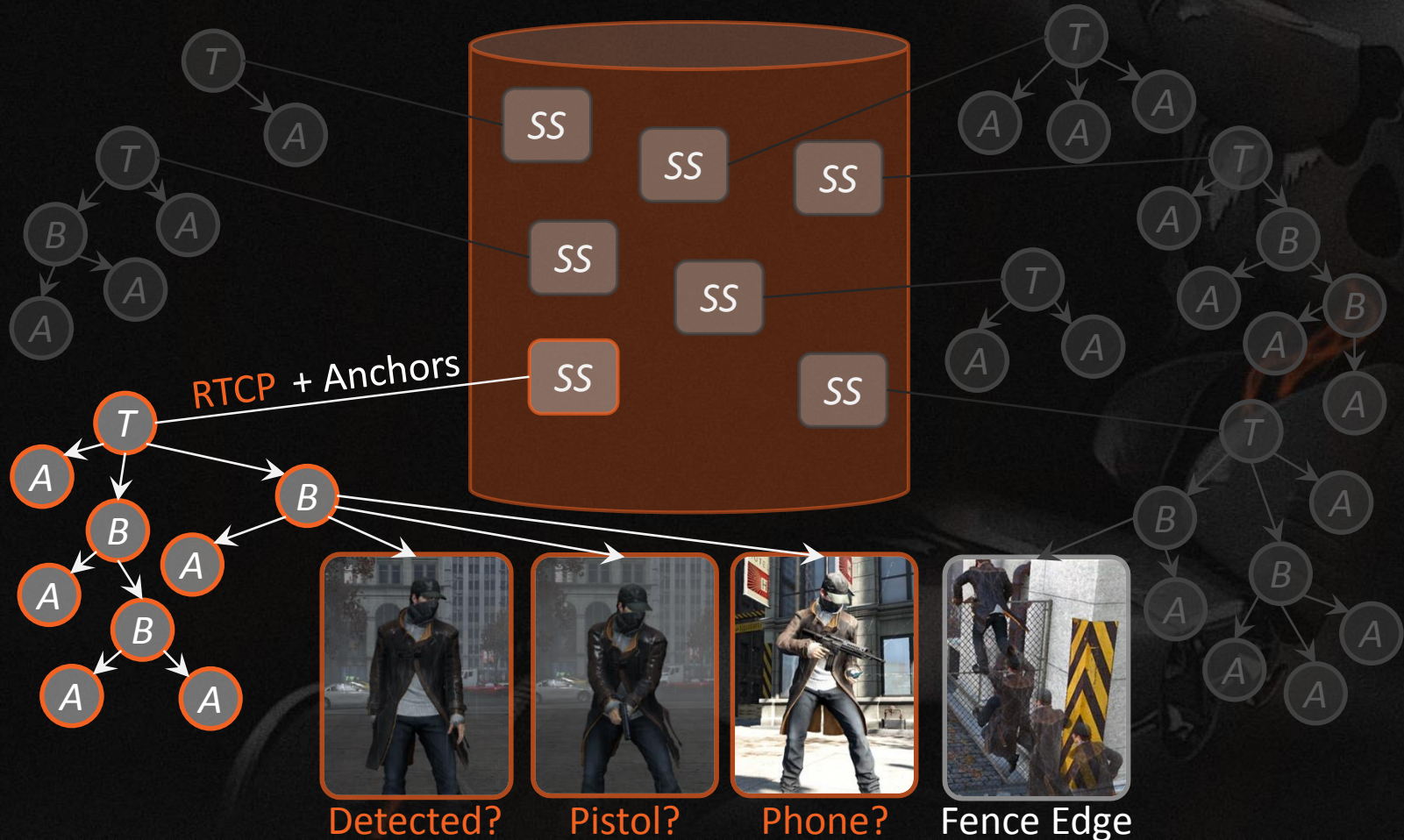


Architecture





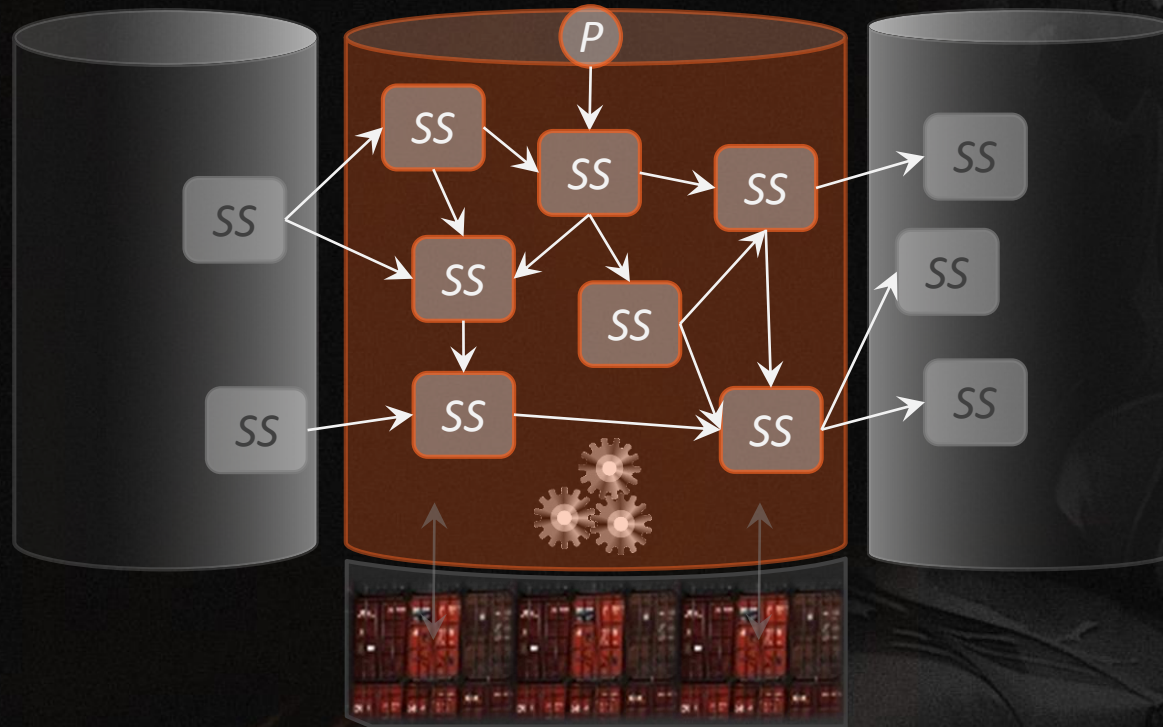
# Sub-State **Dynamic** Animation Trees







# State Composition : Sub-States

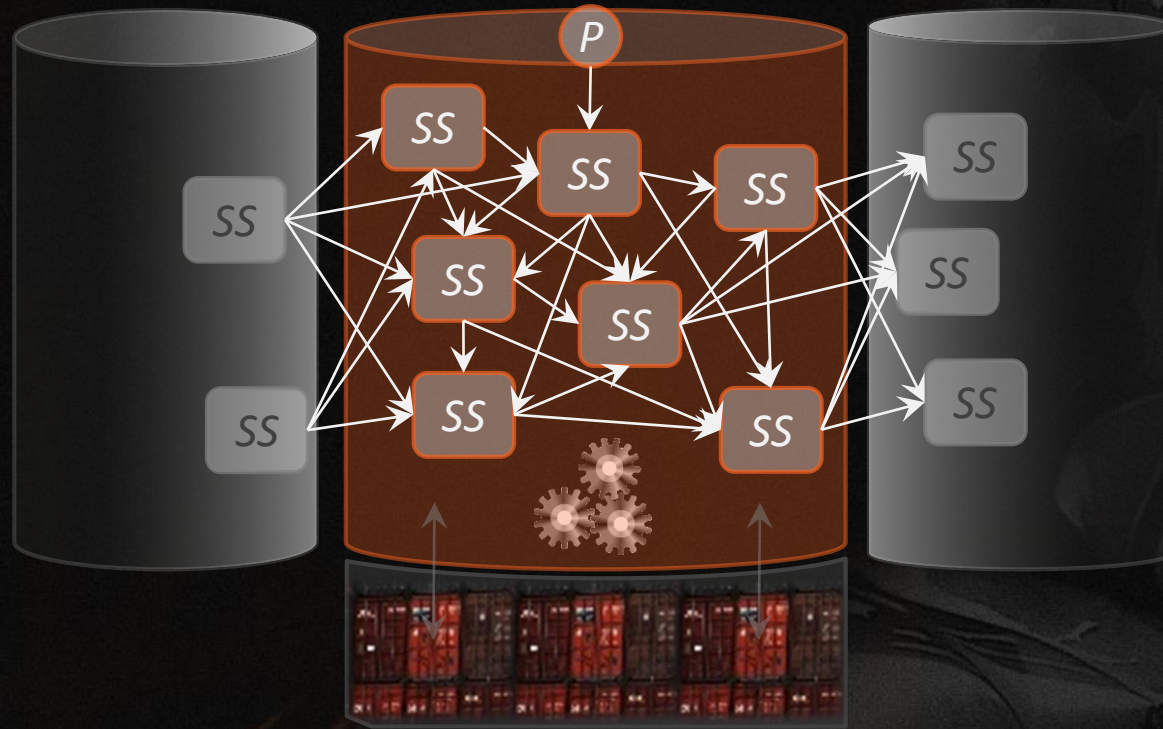


Initially defined transitions...





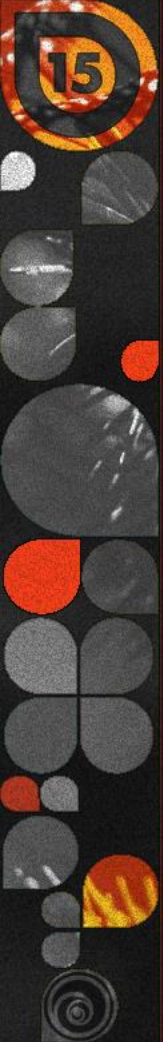
# State Composition : Sub-States



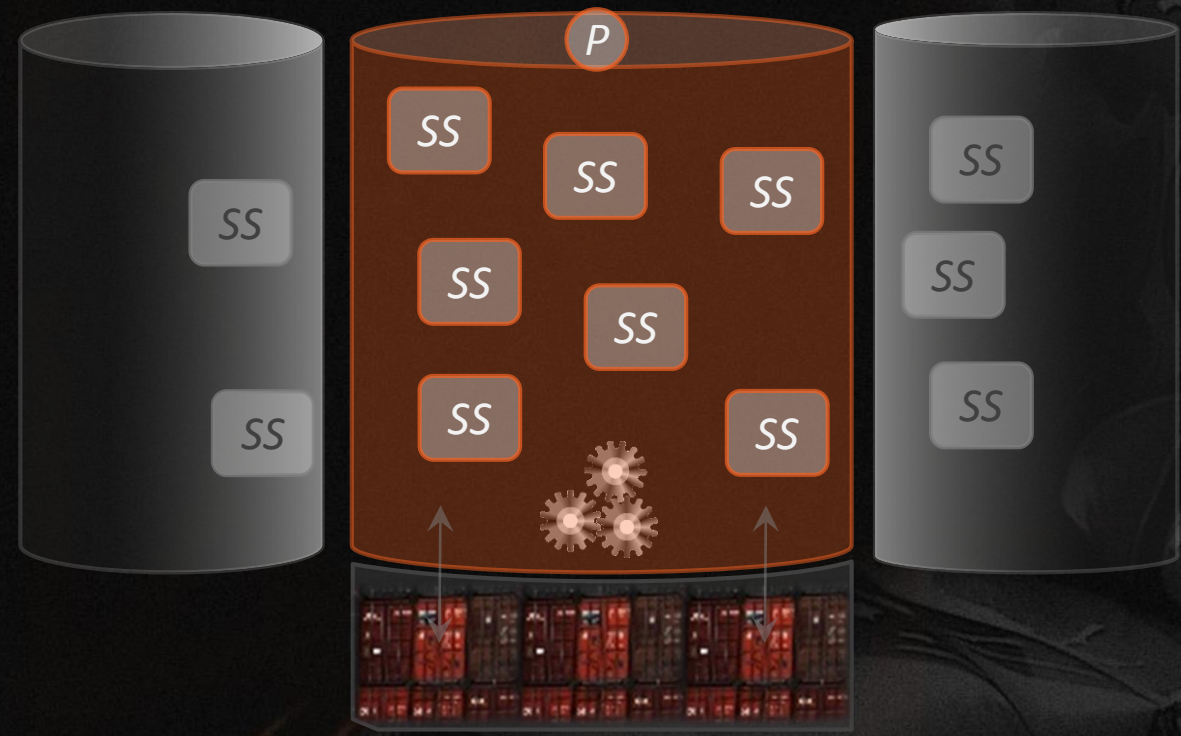
Initially defined transitions...  
Eventually needed transitions..!







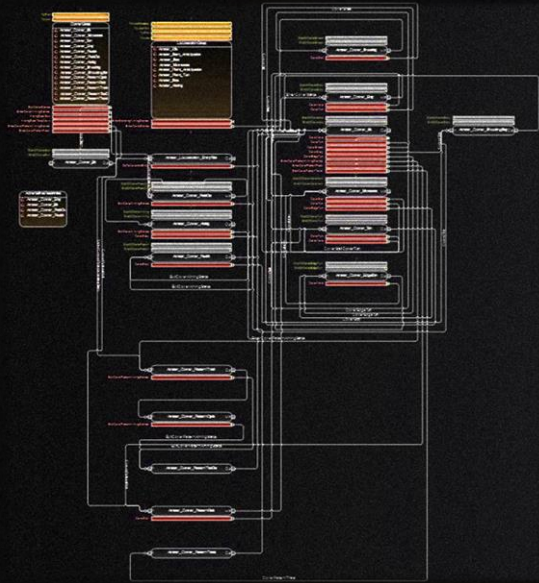
# State Composition : Sub-States



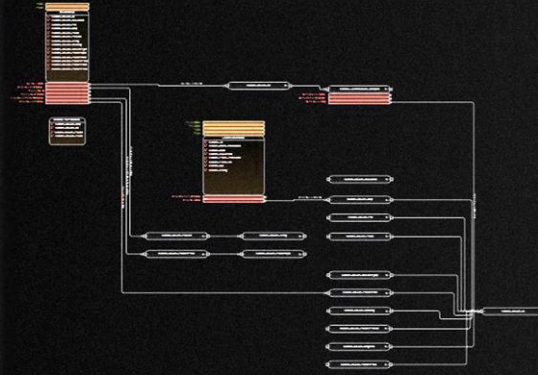
Initially defined transitions...  
Eventually needed transitions..!  
Finally free for all approach



# State Composition : Sub-States



2012



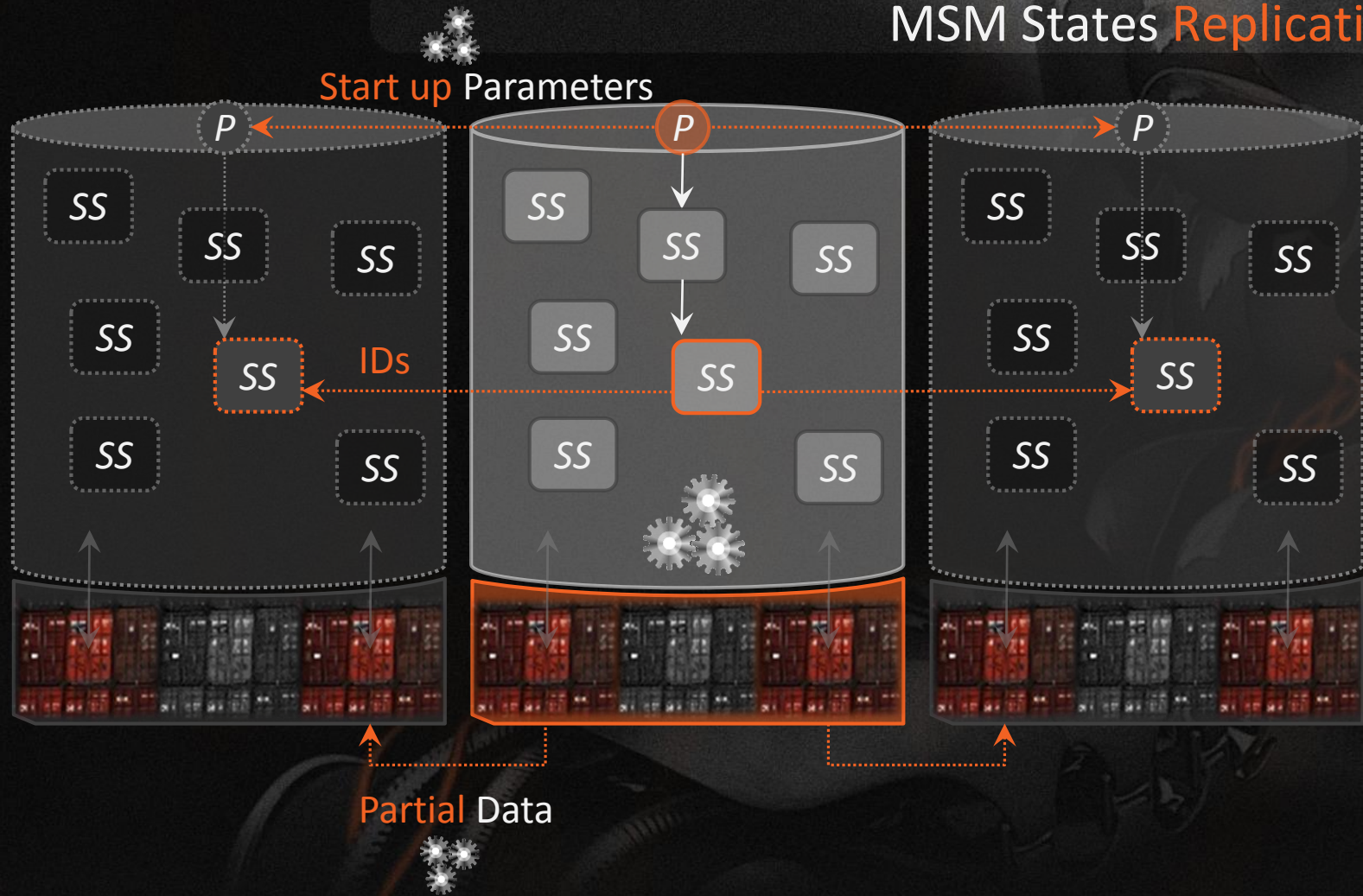
2013

2014





# MSM States Replication





```
class CPlayerCoveredMotionStateDef : public COnFootMotionStateDef
```

```
{  
    DECLARE_STATE_DEF_PARAMS(CPlayerCoveredMotionStateParams);  
  
    DECLARE_STATE_DEF_BEGIN(CPlayerCoveredMotionStateDef, COnFootMotionStateDef, eStatePriority_High)  
        DECLARE_STATE_DEF_SUBSTATE(Approach)  
        DECLARE_STATE_DEF_SUBSTATE(Enter)  
        DECLARE_STATE_DEF_SUBSTATE(Idle)  
        DECLARE_STATE_DEF_SUBSTATE(Start)  
        DECLARE_STATE_DEF_SUBSTATE(Move)  
        DECLARE_STATE_DEF_SUBSTATE(Stop)  
        DECLARE_STATE_DEF_SUBSTATE(TurnAnticipation)  
        DECLARE_STATE_DEF_SUBSTATE(Turn)  
        DECLARE_STATE_DEF_SUBSTATE(RaceTurnTurner)  
    DECLARE_STATE_DEF_END()  
}
```

```
class CPlayerCoveredMotionStateParams : public CBaseStateParams
```

```
{  
    DECLARE_STATE_PARAMS(CPlayerCoveredMotionStateParams, CBaseStateParams)  
  
    DECLARE_NETDATA_BEGIN(CPlayerCoveredMotionStateParams, CBaseStateParams::NetData)  
        DECLARE_NETDATA_MEMBER(m_coverEntryAttachment, CCoverAttachment, CNetDataCoverAttachment)  
        DECLARE_NETDATA_MEMBER(m_randomValue, ndFloat, storm::DataFloat<1, storm::NetDataFloatEpsilon_0_0)  
        DECLARE_NETDATA_MEMBER(m_skipEntry, ndBool, storm::DataBool)  
    DECLARE_NETDATA_END()  
}
```

```
CStateRequest<CPlayerCoveredMotionStateDef> coverRequest;
```

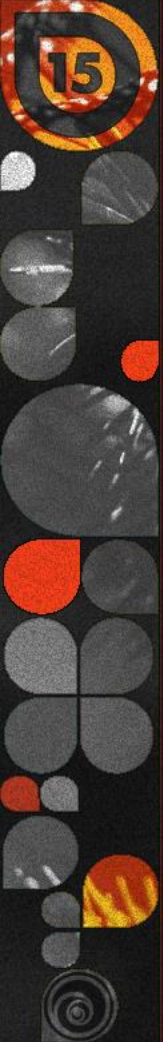
```
coverRequest.GetParams().SetCoverEntryAttachment(coverAttachment);
```

```
coverRequest.GetParams().SetRandomValue(ndRandFloat());
```

```
coverRequest.Send();
```



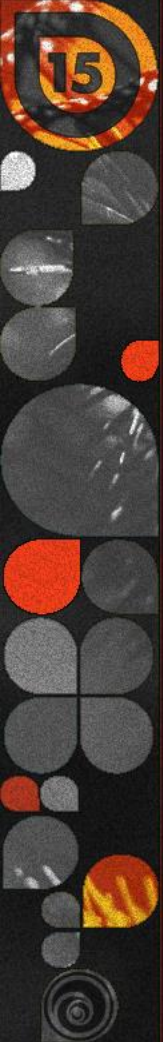




Architecture

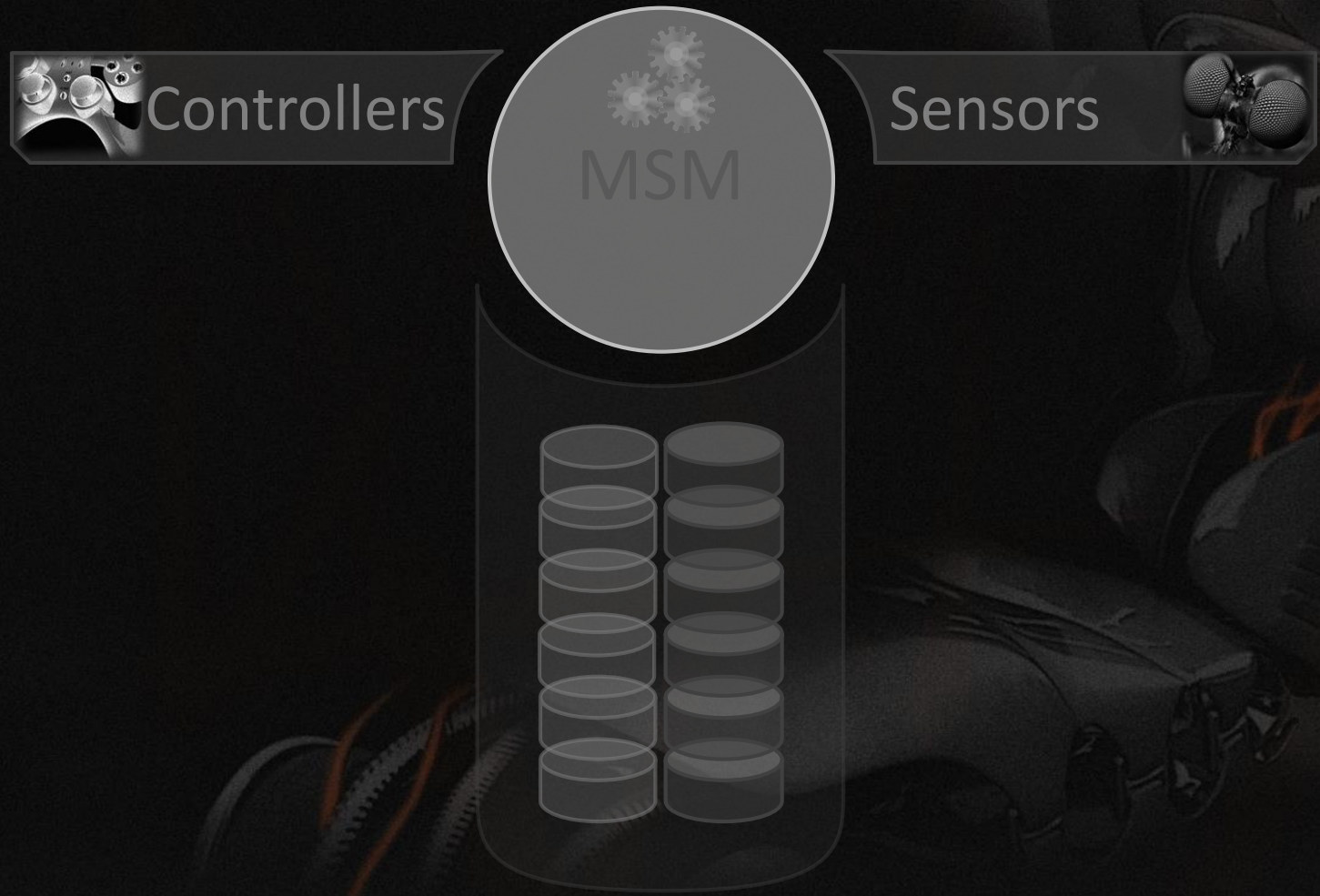


VIDEO EXAMPLE

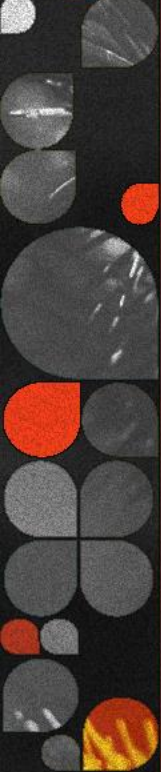


Architecture

# Multitrack State Architecture





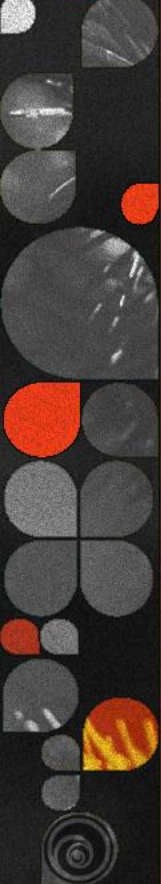


UBISOFT

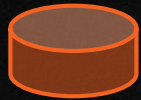
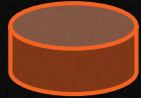
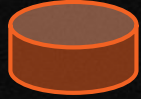
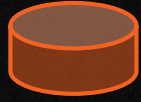
Architecture

The MSM is running 4 tracks





Architecture



The MSM is running 4 tracks

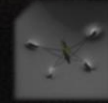






## Motion /'moʊʃən/

ex: *Standing*, *Cover*, *Driving*, *Swimming*



## Action /'æk.ʃən/

ex: *Climbing*, *Reacting*, *Fighting*, *Interacting*



## Stance /stæns/

ex: *Looking*, *Protecting*, *Aiming*



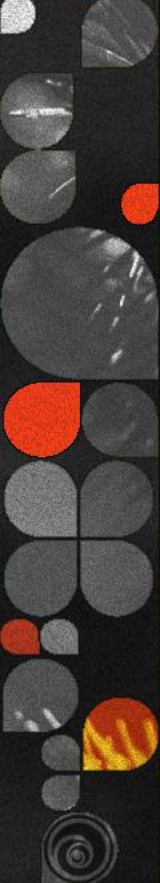
## Kinesic /ki ni:sɪk/

ex: *Firing*, *Throwing*, *Hacking*, *Masking*





Architecture



The MSM is running 4 tracks



**Motion**

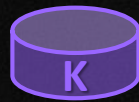
Standing



**Action**

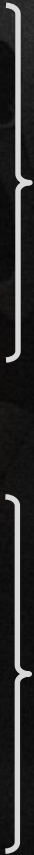


**Stance**



**Kinesic**

Draw







Architecture

The MSM is running 4 tracks



**Motion**

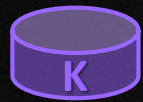
Standing



**Action**



**Stance**



**Kinesic**

Draw

Reload





Architecture

The MSM is running 4 tracks



**Motion**

Standing

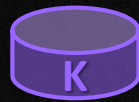


**Action**

Climbing



**Stance**



**Kinesic**

Draw

Reload







The MSM is running 4 tracks



**Motion**

Standing



**Action**

Climbing



**Stance**

Idle



**Kinesic**

Draw

Reload





Architecture

The MSM is running 4 tracks



**Motion**

Standing

Standing



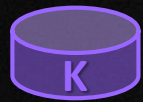
**Action**

Climbing



**Stance**

Idle



**Kinesic**

Draw

Reload



Masking







The MSM is running 4 tracks



**Motion**

Standing

Standing Goto



**Action**

Climbing

Use ATM



**Stance**

Idle



**Kinesic**

Draw

Reload



Masking



The MSM is running 4 tracks



**Motion**

Standing

Standing Goto



**Action**

Climbing

Use ATM



**Stance**

Idle

Idle



**Kinesic**

Draw

Reload



Masking





Architecture

The MSM is running 4 tracks



**Motion**

Standing

Standing



**Action**

Climbing

Use ATM



**Stance**

Idle

Idle

Look At



**Kinetic**

Draw

Reload

Masking





The MSM is running 4 tracks



**Motion**

Standing

Standing

Standing



**Action**

Climbing

Use ATM



**Stance**

Idle

Idle

Look At



**Kinesic**

Draw

Reload

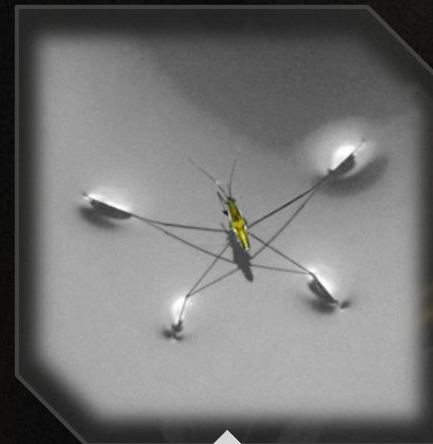


Masking





Seamless **Fluidity** Realism  
Consciousness  
Diversity Contextual  
**Precision**  
Connectivity



Game Feel

Usability

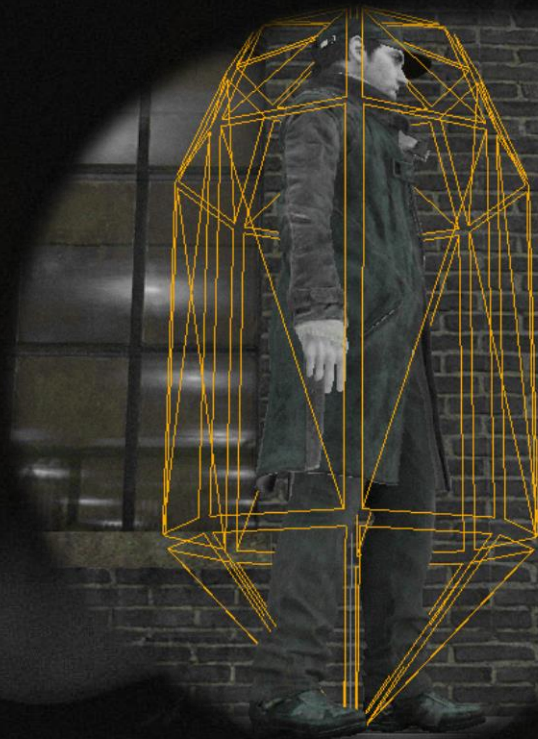
Grounding

Spatial Awareness

Body Awareness

## Animation driving displacement

- mostly translations
- sometimes rotations





## Animation driving displacement

- mostly translations
- sometimes rotations

## Skeleton smoothed collision

- ~~cylinder shape~~ (no auto step)
- ~~capsule shape~~ (no high angle step)
- pencil shape



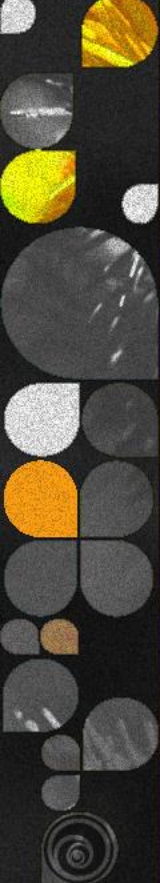
## Heading is the intention

- expressed by head direction
- read from angular input
- high frequency filtering

## Facing is body lagging







Motion

Instant head direction intention

VIDEO EXAMPLE



UBISOFT

# Displacement **sliding** for responsive **precision**

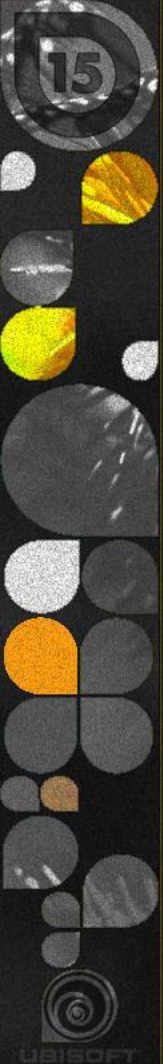
**Facing** is body lagging

- expressed by shoulders direction
- read from lateral input
- high frequency filtering

**Displacement** is sliding **input** heading

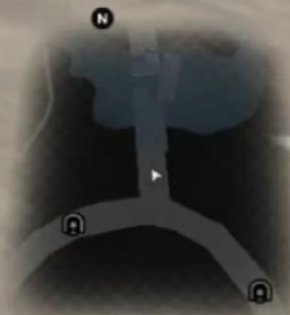
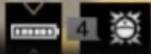






Motion

Instant shoulders direction intention



UBISOFT

- Custom entry in animations
- Adaptive pose matching
- Latency reduction

*Movement :*

LEFT FRONT



RIGHT PASS



RIGHT FRONT

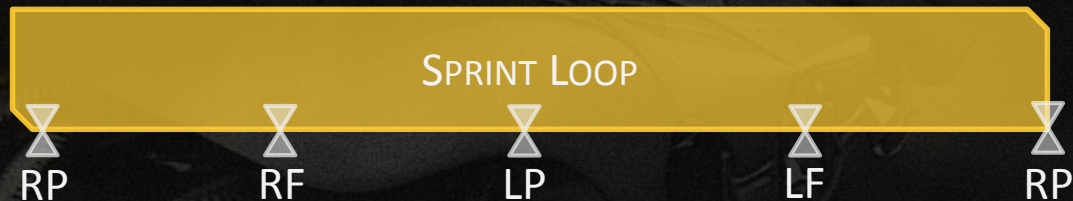


LEFT PASS

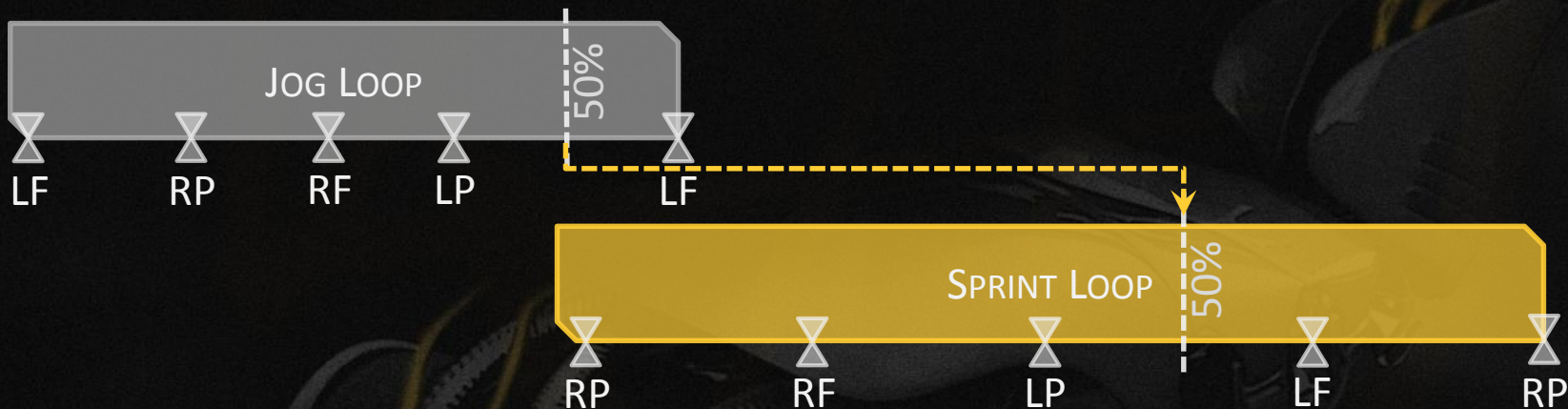




- Custom entry in animations
- Adaptive pose matching
- Latency reduction

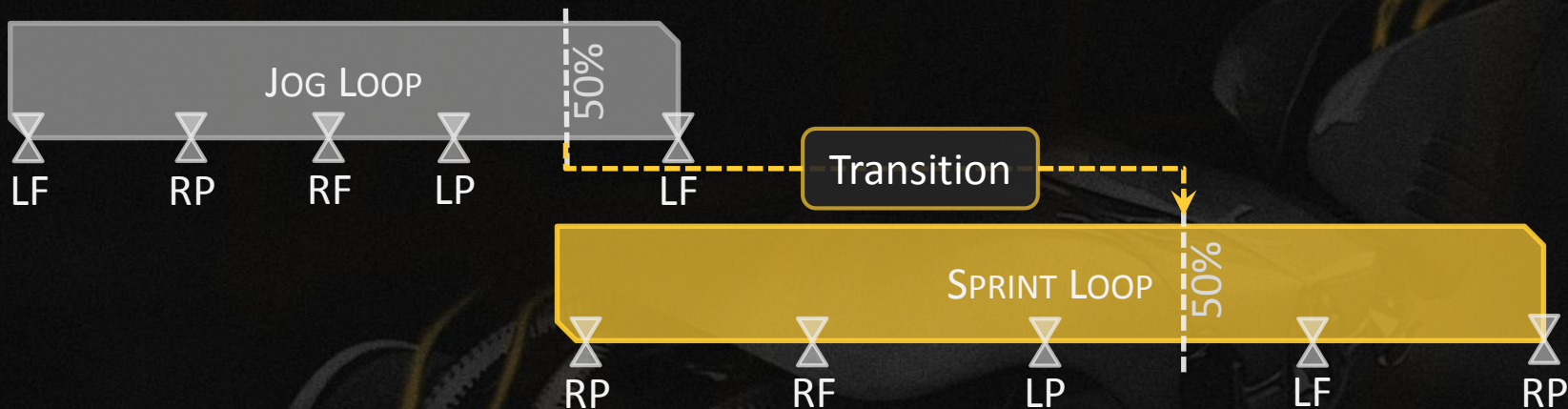


- Custom entry in animations
- Adaptive pose matching
- Latency reduction

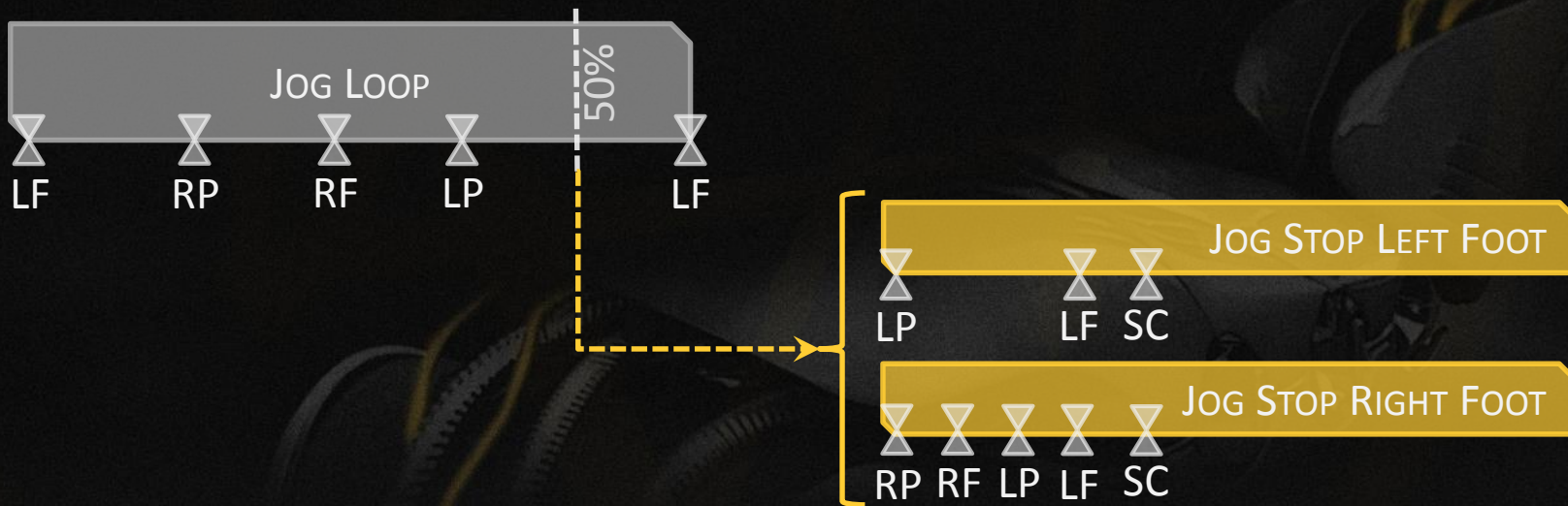




- Custom entry in animations
- Adaptive pose matching
- Latency reduction

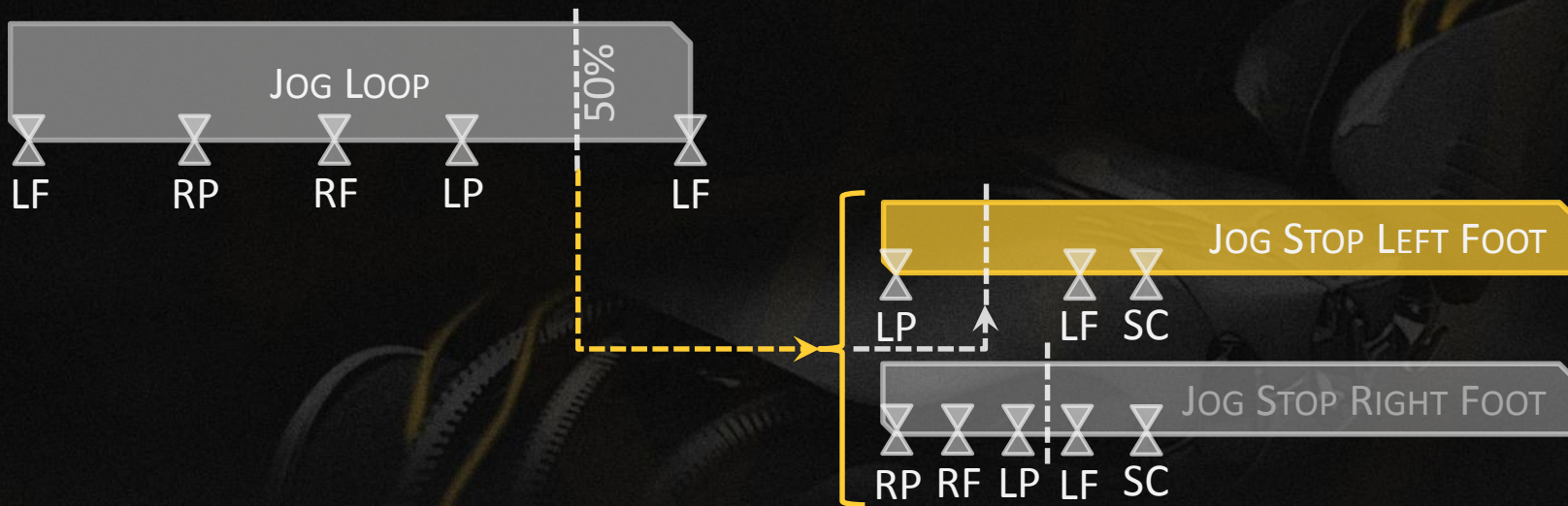


- Custom entry in animations
- Adaptive pose matching
- Latency reduction

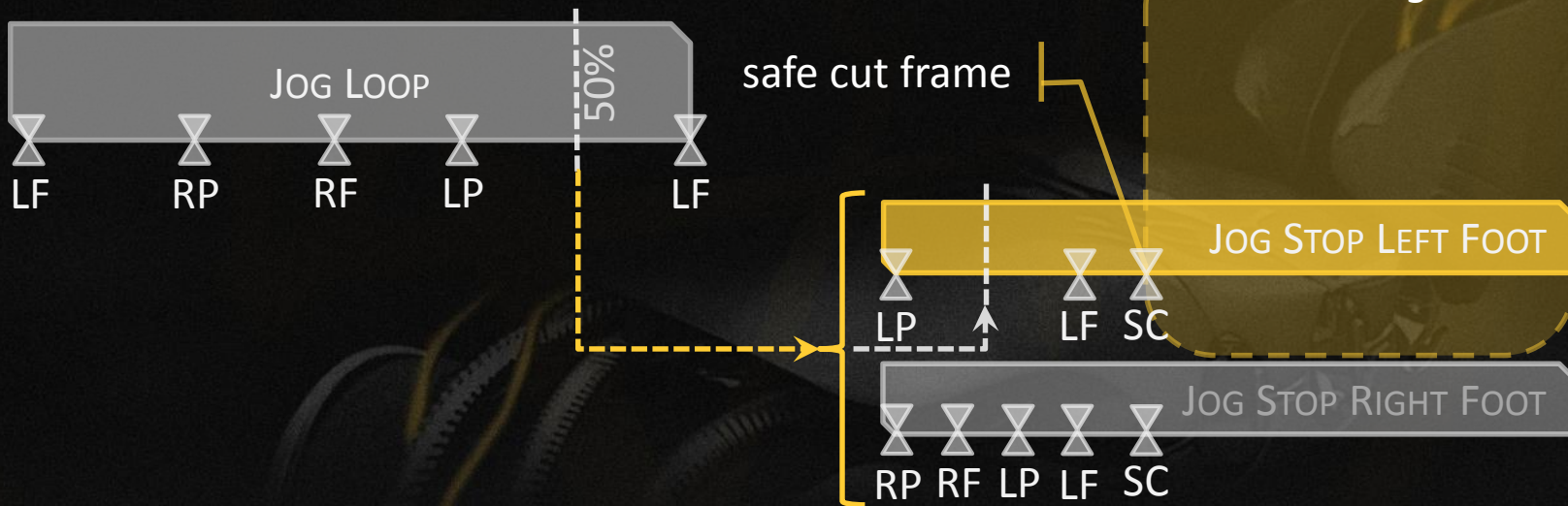




- Custom entry in animations
- Adaptive pose matching
- Latency reduction



- Custom entry in animations
- Adaptive pose matching
- Latency reduction







Motion

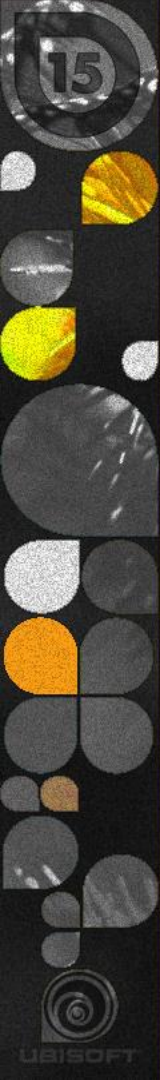


Pose **matching** transitions navigation

VIDEO EXAMPLE



UBISOFT



- Optical illusion : Head motion reactivity
- Read mind for direction & speed : Fixed delay
- Anticipation of ~6@8 frames (*playtest*)

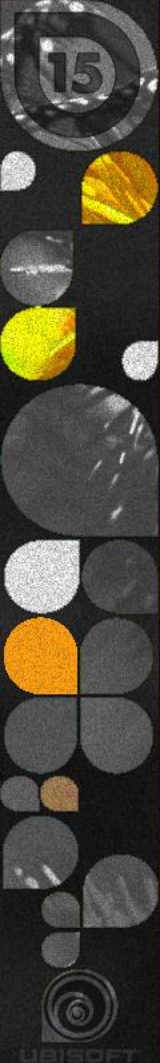
START

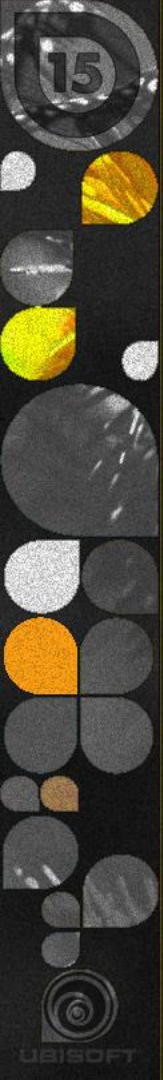


- Optical illusion : Head motion reactivity
- Read mind for direction & speed : Fixed delay
- Anticipation of ~6@8 frames (*playtest*)
- Same philosophy applied to plants & turns

ANTICIPATION  
(*common*)

TRANSITION  
to *idle*, to *walk*, to *jog*, to *sprint*





Motion

Responsive starts/plants for precise navigation



VIDEO EXAMPLE

UBISOFT



Consciousness  
Seamless  
Realism  
Connectivity  
Contextual  
Fluidity  
Diversity  
Precision



Game Feel

Usability

Grounding

Spatial Awareness

Body Awareness



## Dedicated sensors for contextual climbing

### Physical **shape/ray** sensor

- Real-time & Functionnal, but...
- Expensive on many characters

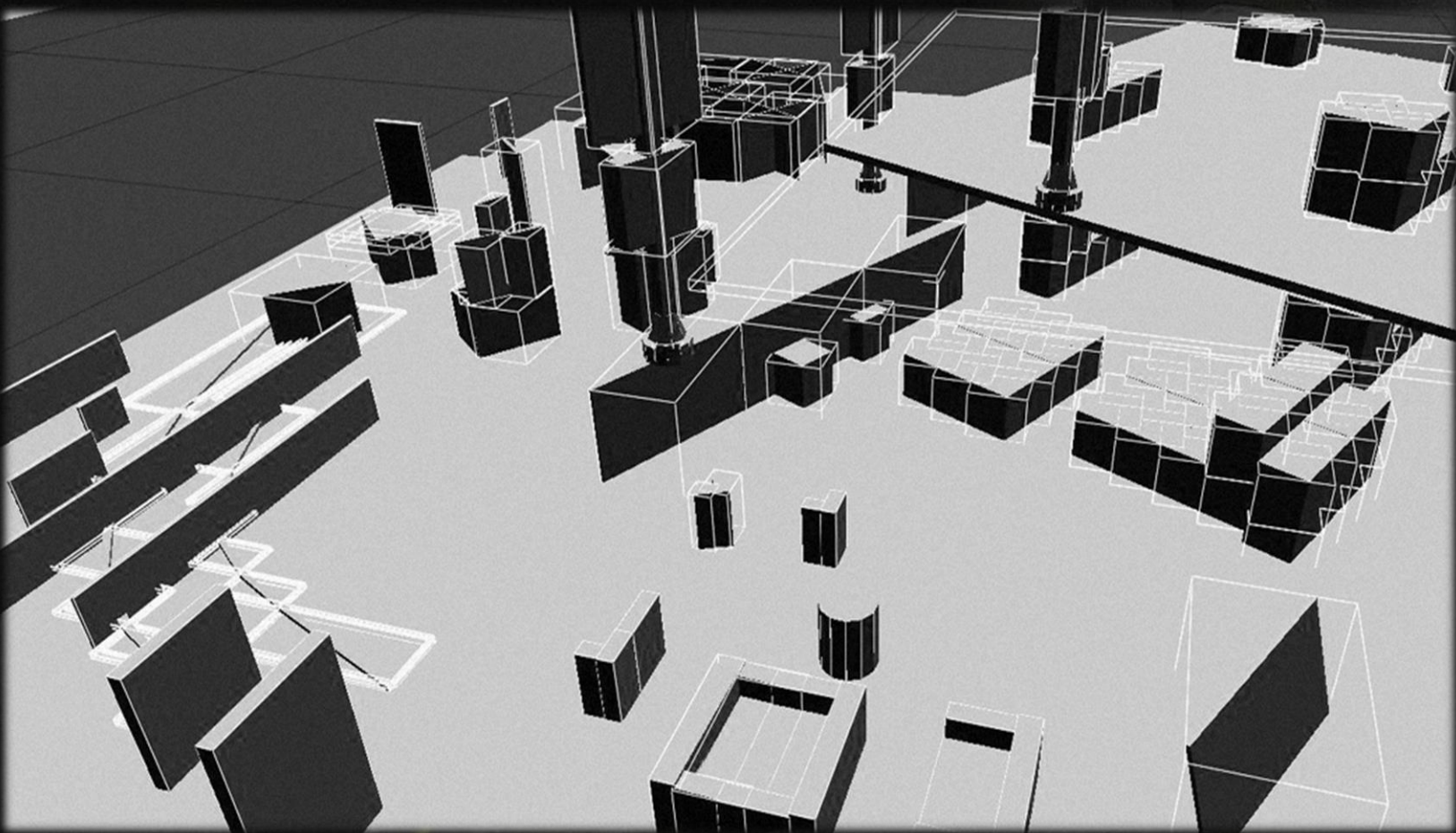
### Edge **annotation** sensor

- Offline generated & “Precise”
- Cheap with some filtering





# Collision meshes [Annotations generation]

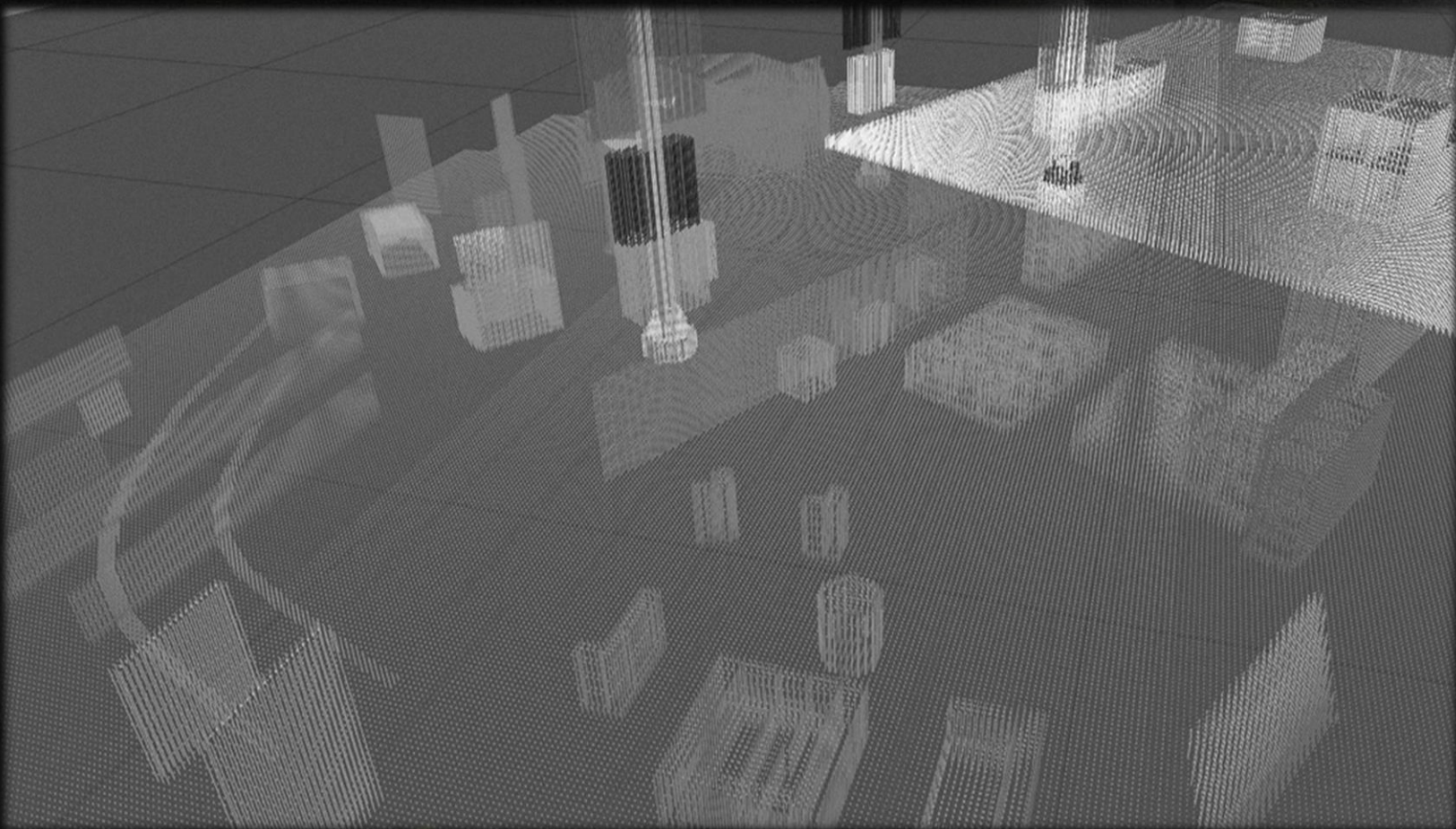




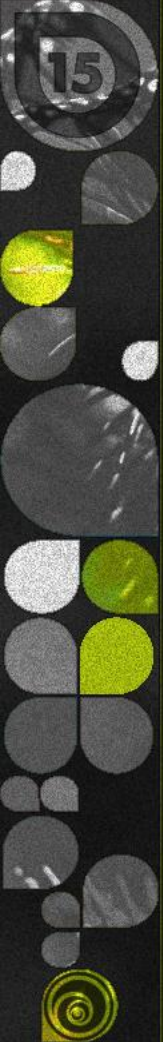
Action



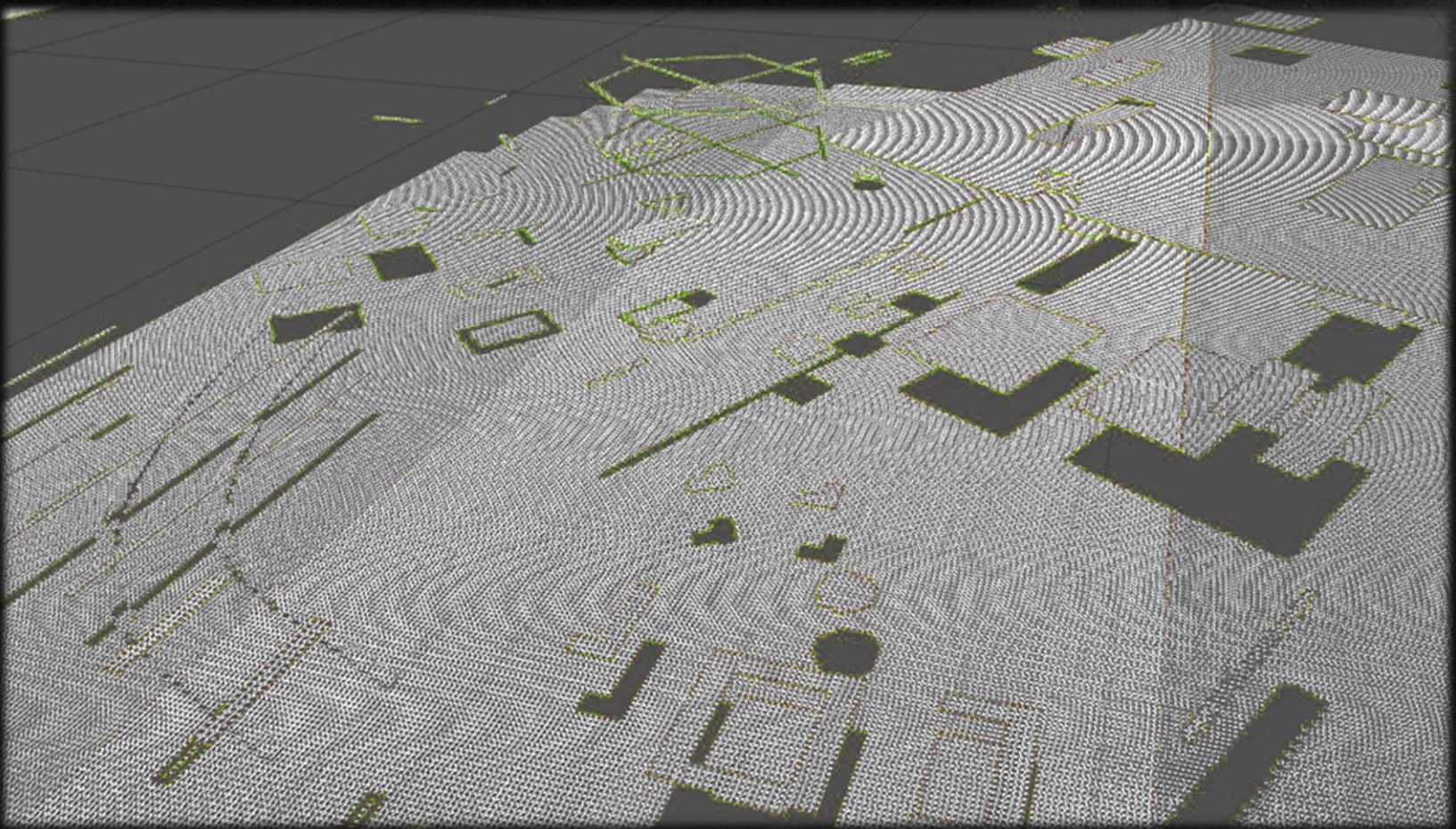
# Voxel height field [Annotations generation]





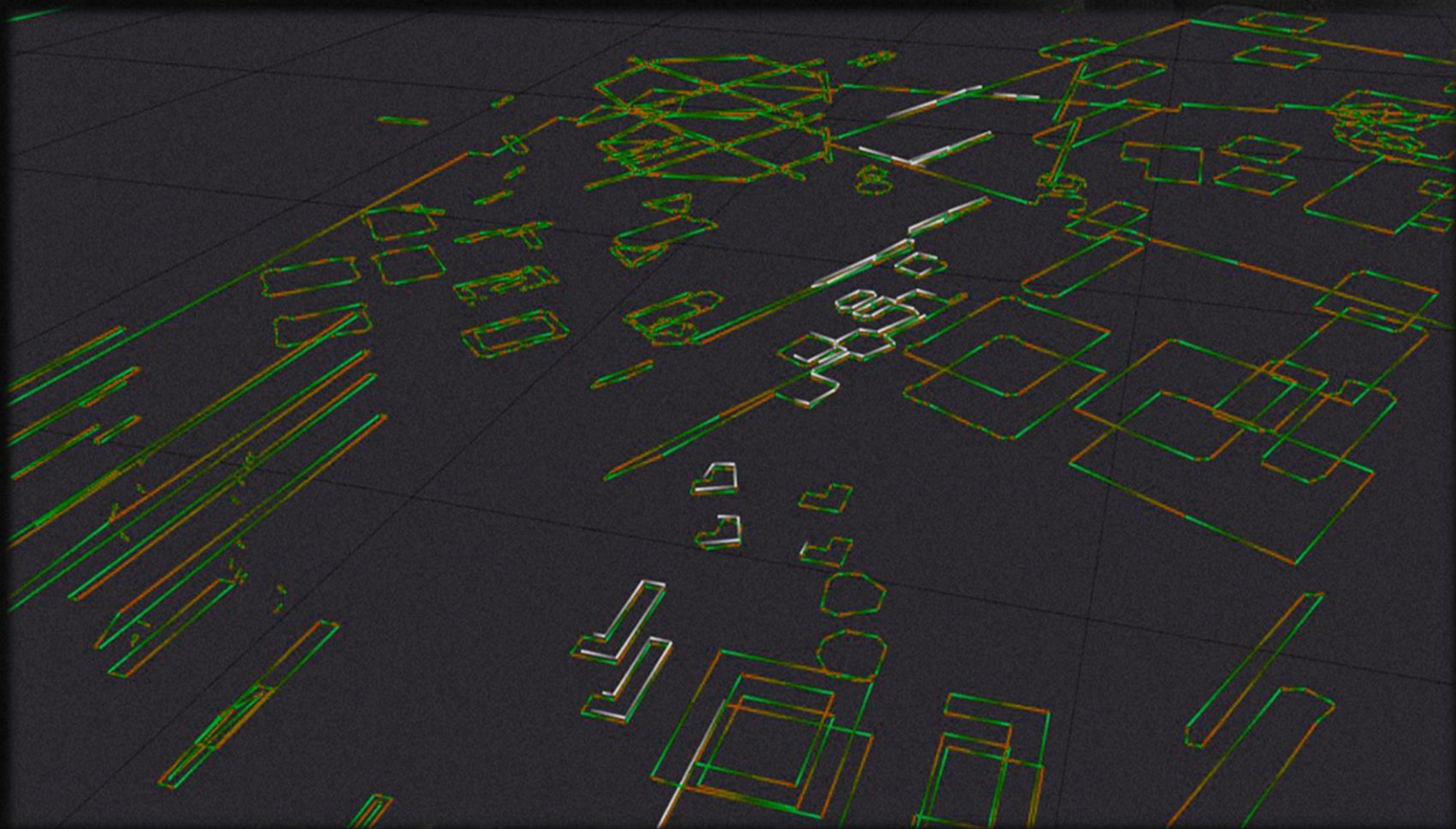


# Surface detection [Annotations generation]



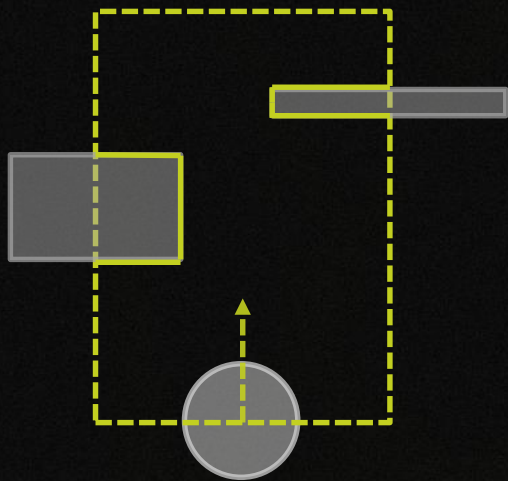


# Object contour [Annotations generation]





# Climbing interaction contextual scanning

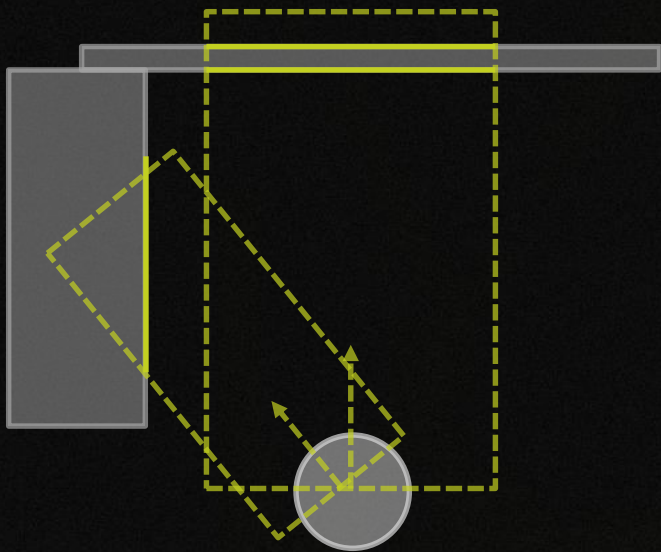


## Volumetric casts

- Multiple detection shapes



# Climbing interaction contextual scanning



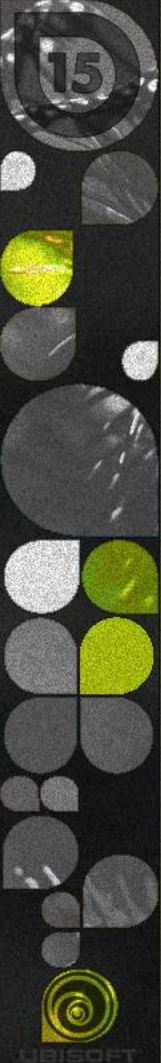
## Multiple scans

- Enhanced environment detection





# VIDEO EXAMPLE



Action



# Data **segmentation** for **seamless** sequence

- One single mocap clip : Seamless action
- One single animation scene : Easier editing
- Multiple sub-states logic for scalability

CLIMBING ON

*Original animation*





15

Action

# Data segmentation for seamless sequence

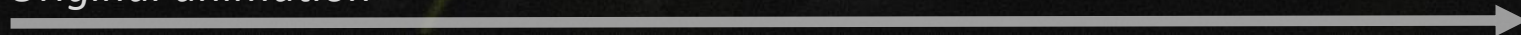


TRAN  
IN

BODY

TRAN  
OUT

*Original animation*

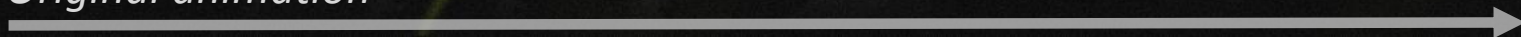


UBISOFT

## Data markup for seamless sequence



*Original animation*

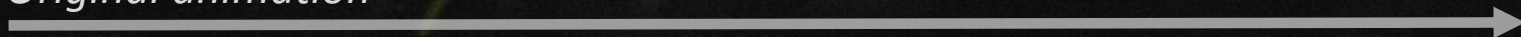




15

Action

## Data markup for seamless sequence

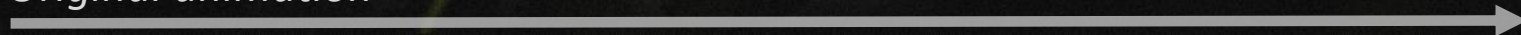
*Original animation*

UBISOFT

15

Action

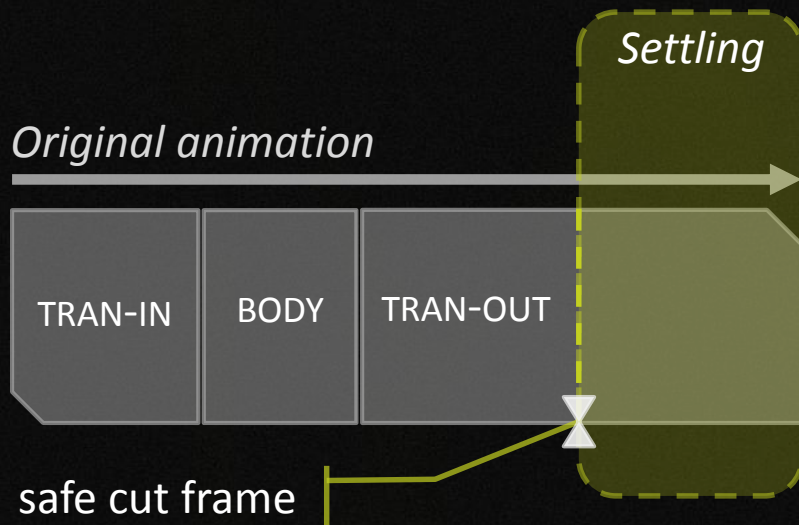
## Data markup for seamless sequence

*Original animation*

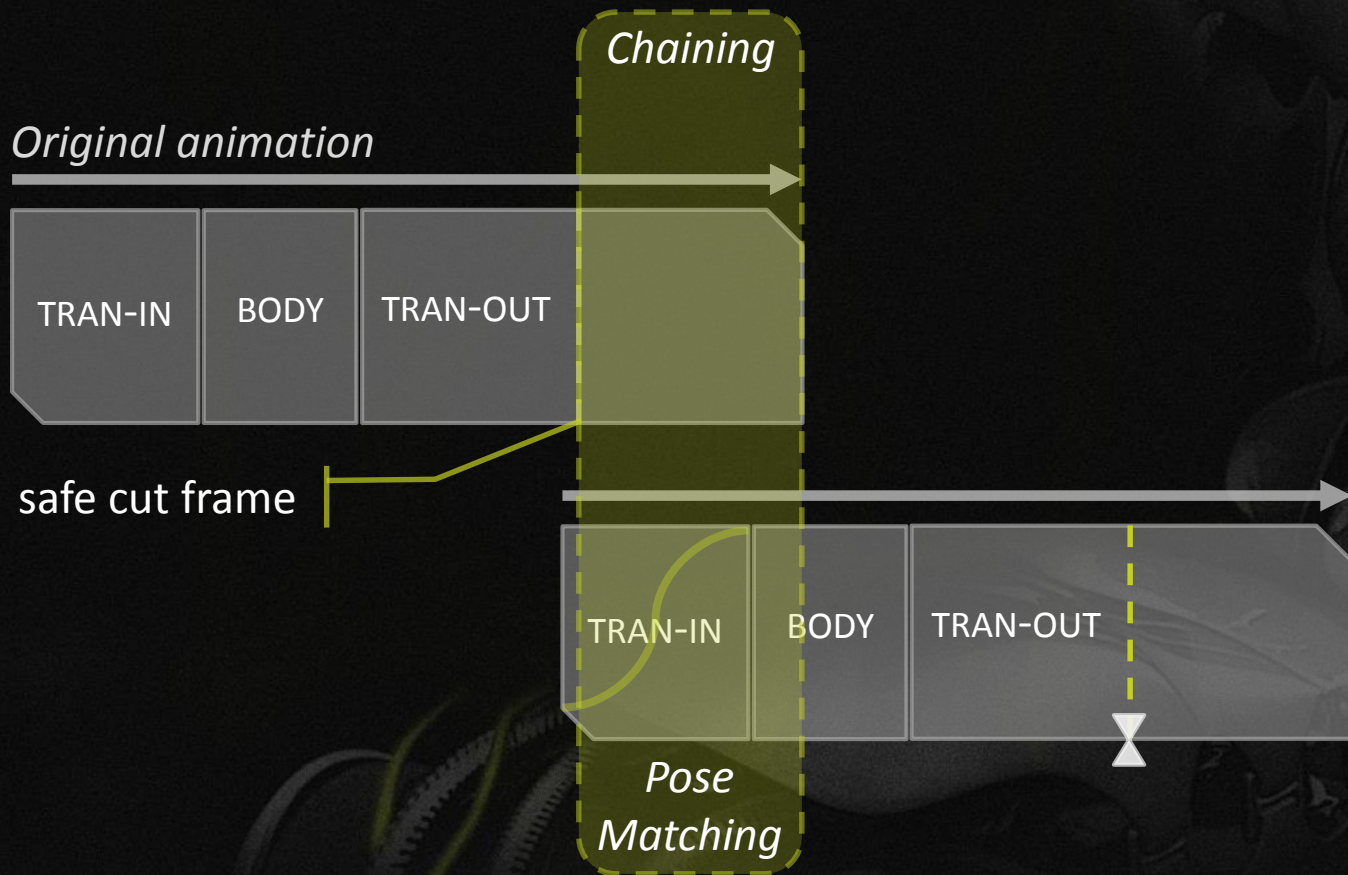
UBISOFT



## Data markup for seamless sequence



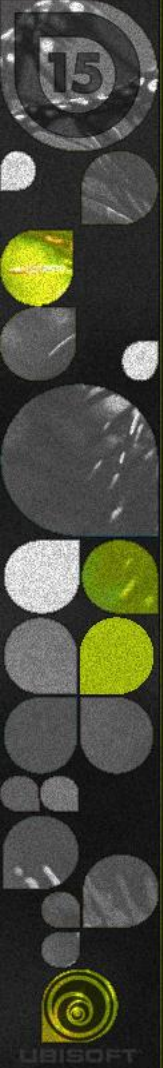
## Data markup for seamless sequence



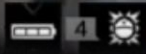


Seamless sequence with interruptions

# VIDEO EXAMPLE



Action



UBISOFT

# Dedicated sensors for contextual reactions

## Stimulus sensor

- Catching impacts, explosions, etc...
- Managing reactions & impulses

## Body sensor

- Acquiring bone poses
- Computing body directions





Presets for required animations  
Custom data for each ragdoll bone



Bone Settings

Stiffness  
Collision  
...

Rigidbody Settings

Max Linear velocity  
Max Angular velocity  
Velocity Damping  
...



15

Action

## Common ragdoll "organic" simulation



UBISOFT



15

Action

# Simulation for seamless realization



BALLISTIC

FLYING

LANDING

GET  
UP



UBISOFT

15

Action

# Simulation for seamless realization



BALLISTIC

FLYING

LANDING

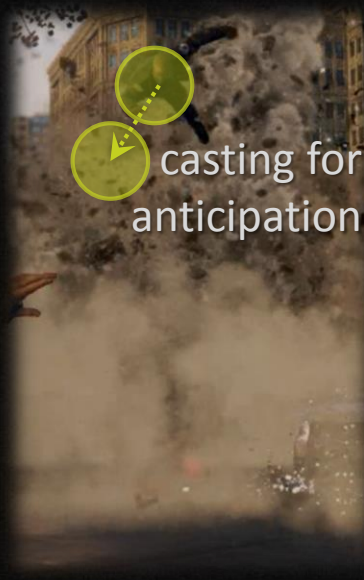
GET  
UP



UBISOFT



## Simulation for seamless realization



BALLISTIC

FLYING

LANDING

GET  
UP

15

Action

# Simulation for seamless realization



release

BALLISTIC

FLYING

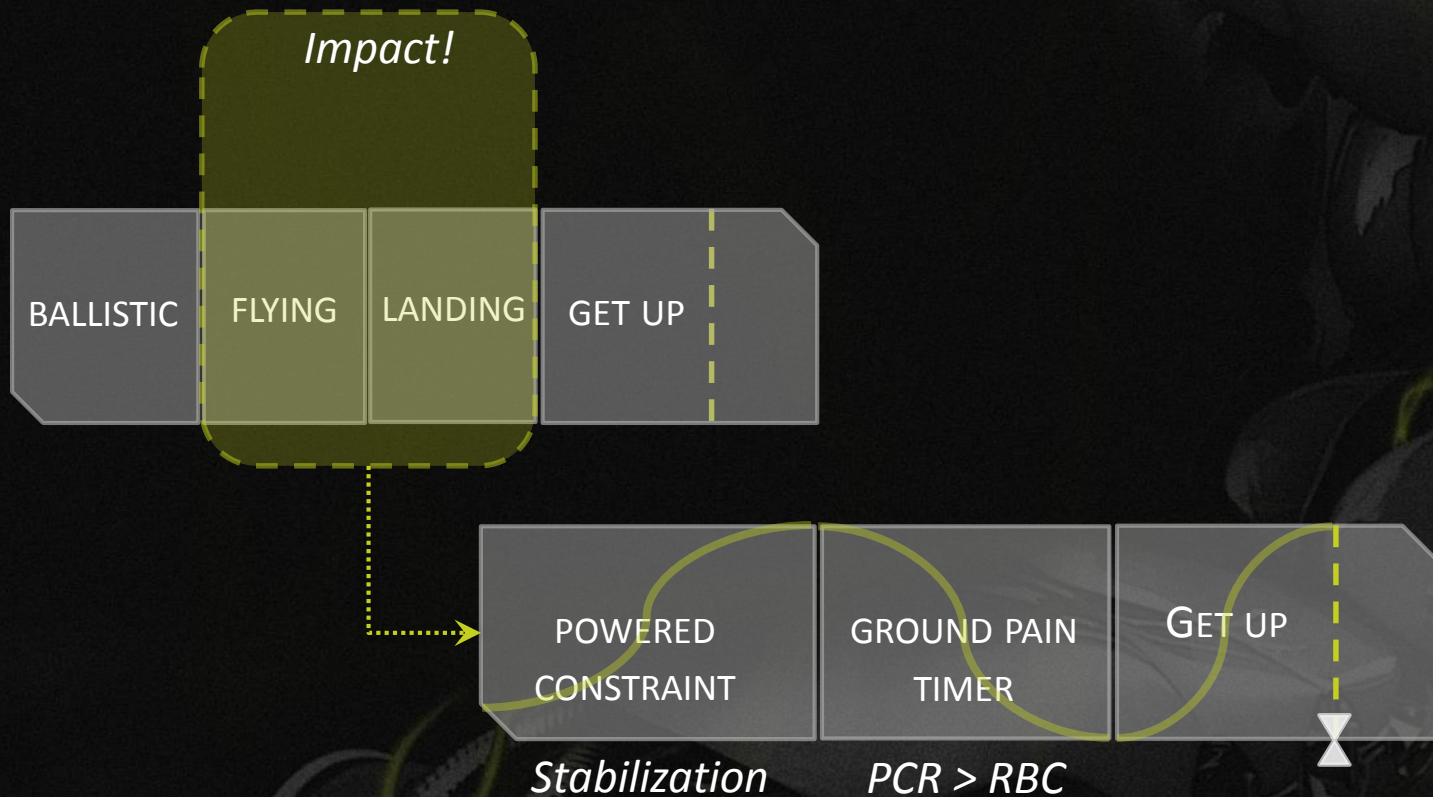
LANDING

GET  
UP



UBISOFT



Impact anticipation for **seamless** ragdoll blend

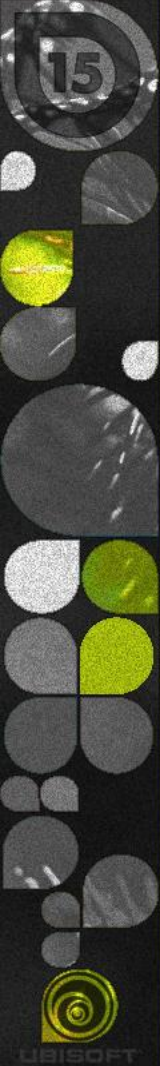
15

Action

Bike ejection **trajectory** simulation

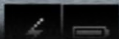
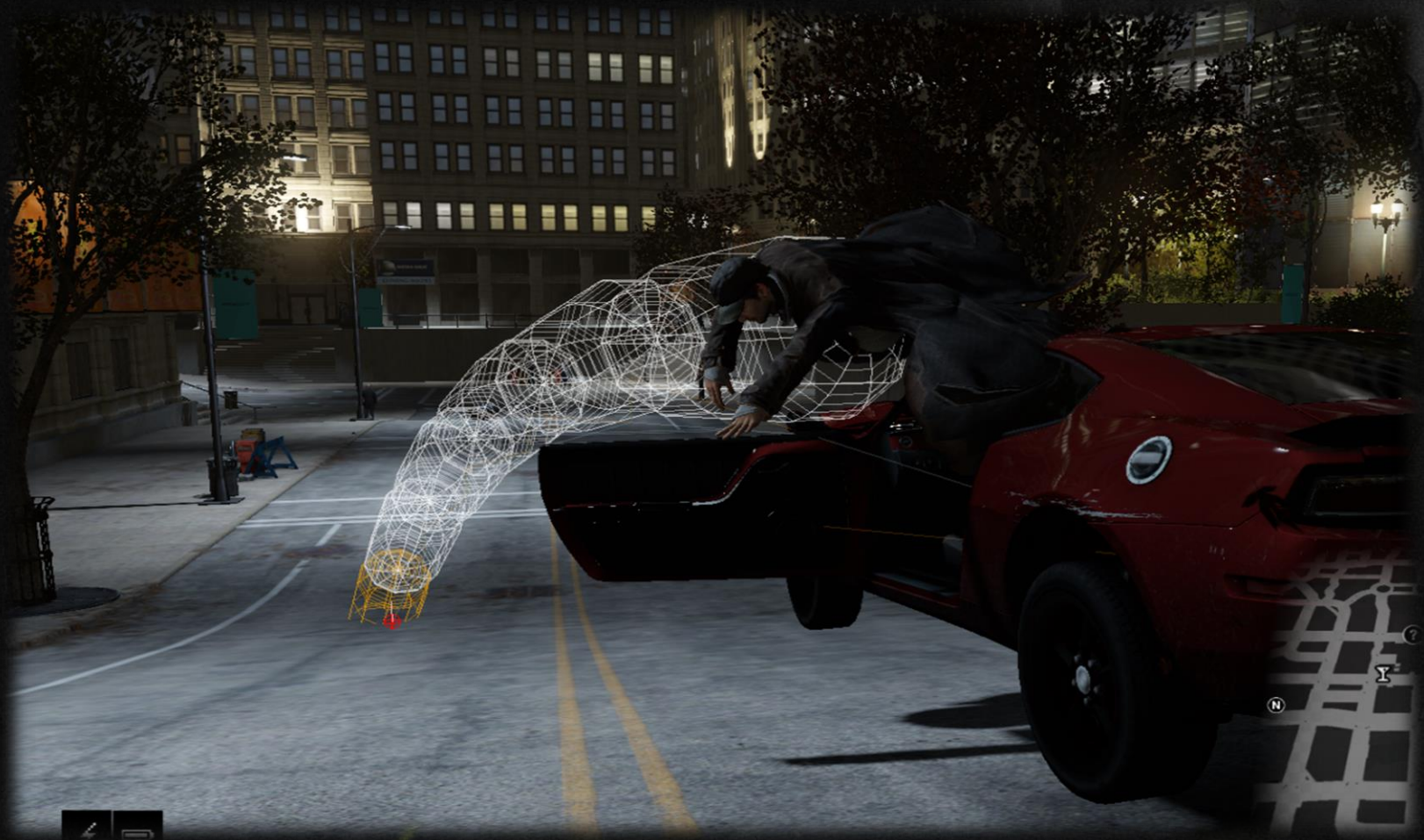
UBISOFT





Action

# Car ejection **trajectory** simulation



N

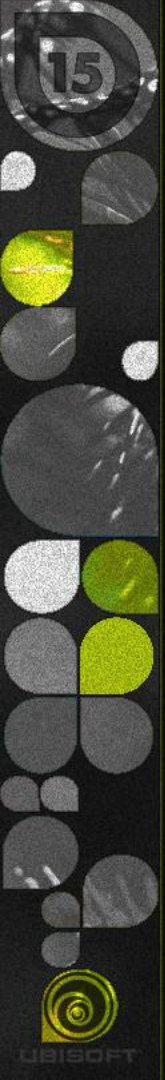
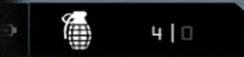
Y

?

UBISOFT

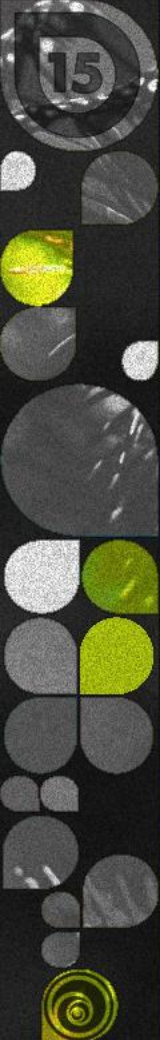
# Explosion ejection **trajectory** simulation

Action



UBISOFT





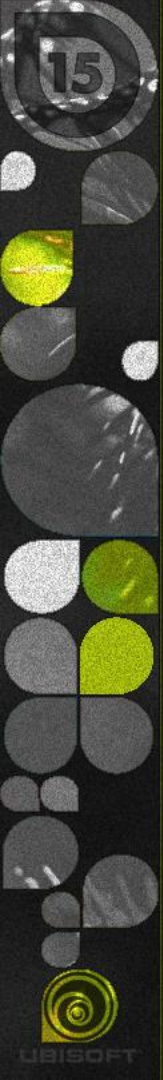
Action

# Simulation for contextual realization



Seamless sequence with interruptions

# VIDEO EXAMPLE



Action



Realism  
Diversity  
Precision  
Seamless  
Connectivity  
Consciousness  
Contextual  
Fluidity



Game Feel

Usability

Grounding

Spatial Awareness

Body Awareness



## Characters changing stance



object in hands

context

mood

Emphasize emotion diversity

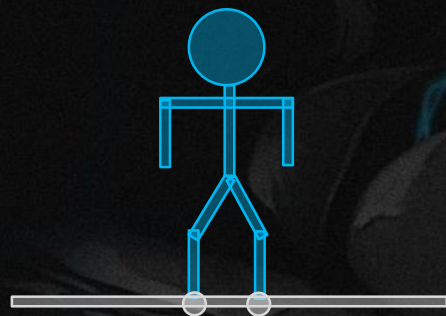
Break idle stillness





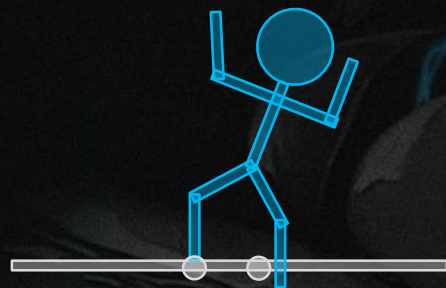
## Additive layering on full body

- different weights for each bone
- bone weights different for each speed
- feet IK resolution post layering
- ...not only useful for slopes!



## Additive layering on full body

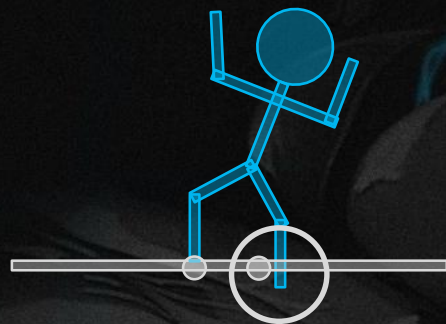
- different weights for each bone
- bone weights different for each speed
- feet IK resolution post layering
- ...not only useful for slopes!





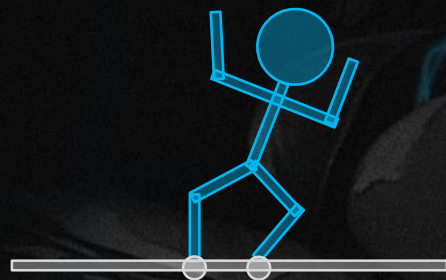
## Additive layering on full body

- different weights for each bone
- bone weights different for each speed
- feet IK resolution post layering
- ...not only useful for slopes!



## Additive layering on full body

- different weights for each bone
- bone weights different for each speed
- feet IK resolution post layering
- ...not only useful for slopes!





## Additive layering on full body

- different weights for each bone
- bone weights different for each speed
- feet IK resolution post layering
- ...not only useful for slopes!



## Previous procedural approaches

- Fully in model space
- Fully in local space
- Blend of model & local space
- Cried...Decided to go data driven with custom spaces





# Data driven look-at layering for consciousness



3D look at point

FEET BLEND SPACE COMPUTING

INJECT TORSO PITCH/YAW RTCP

UPPERBACK BLEND SPACE COMPUTING

INJECT NECK PITCH/YAW RTCP

INJECT HEAD PITCH/YAW RTCP

EYES COMPENSATION

# Data driven look-at layering for consciousness



3D look at point

FEET BLEND SPACE COMPUTING

INJECT TORSO PITCH/YAW RTCP

UPPERBACK BLEND SPACE COMPUTING

INJECT NECK PITCH/YAW RTCP

INJECT HEAD PITCH/YAW RTCP

EYES COMPENSATION



# Data driven look-at layering for consciousness



3D look at point

FEET BLEND SPACE COMPUTING

INJECT TORSO PITCH/YAW RTCP

UPPERBACK BLEND SPACE COMPUTING

INJECT NECK PITCH/YAW RTCP

INJECT HEAD PITCH/YAW RTCP

EYES COMPENSATION



## FEET BLEND SPACE COMPUTING

```
// Get current referential
CQuaternion CRightFoot( sourceResult->m_displacementTransform.GetRotation()
* (modelCommonRaw->GetModelTransform( rightFootIdx, skeletonRaw, jointTransformArray ).GetRotation() * InvTSRightFoot) );

CQuaternion CLeftFoot( sourceResult->m_displacementTransform.GetRotation()
* (modelCommonRaw->GetModelTransform( leftFootIdx, skeletonRaw, jointTransformArray ).GetRotation() * InvTSLeftFoot) );

CQuaternion feetRot;
feetRot.QuaternionSlerp( CRightFoot, CLeftFoot, 0.5f );
feetRot.RemoveAllExceptAxis( CVector3( 0.f, 0.f, 1.f ) );
```

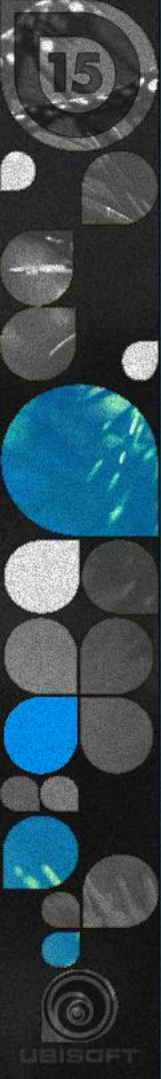
## UPPERBACK BLEND SPACE COMPUTING

```
// Get current referential
ndQuat upperBack( sourceResult->m_displacementTransform.GetRotation()
* (modelCommonRaw->GetModelTransform( upperBackIdx, skeletonRaw, jointTransformArray ).GetRotation() * InvTSUpperBack) );

m_lookatWorldTransform = ndPosQuatTransform( upperBack, neckPos );
```



# VIDEO EXAMPLE

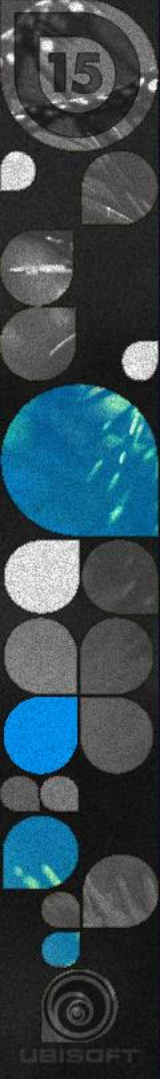


Stance



## Contextual data scalability

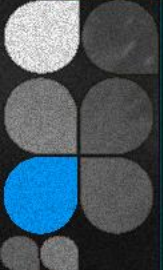
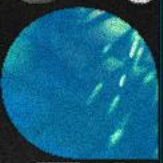
- distinction between glance & stare
- in car head look-at recycling
- anti-look-at for fire protection
- sky is the limit...  
(within animator & memory constraints)







Stance



Data driven reaction layered system

# VIDEO EXAMPLE



Precision  
Diversity  
Seamless  
Contextual  
Fluidity  
Consciousness

# Realism

# Connectivity



Game Feel

Usability

Grounding

Spatial Awareness

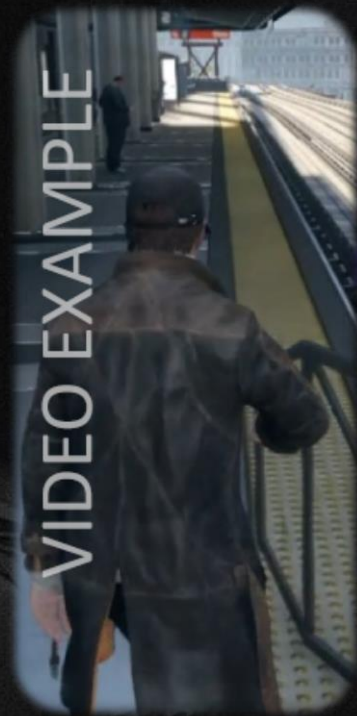
Body Awareness





## Enhance the power of touching

- pushing doors
- grabbing grenades
- putting on the mask
- getting hands in pockets
- ...or even say hello to strangers!



## Multiple recoil layering for realism



WEAPON MECHANISM MOTION

WEAPON SKELETON MOTION

FULL BODY MOTION ADDITIVE

CAMERA GAMEPLAY MOTION

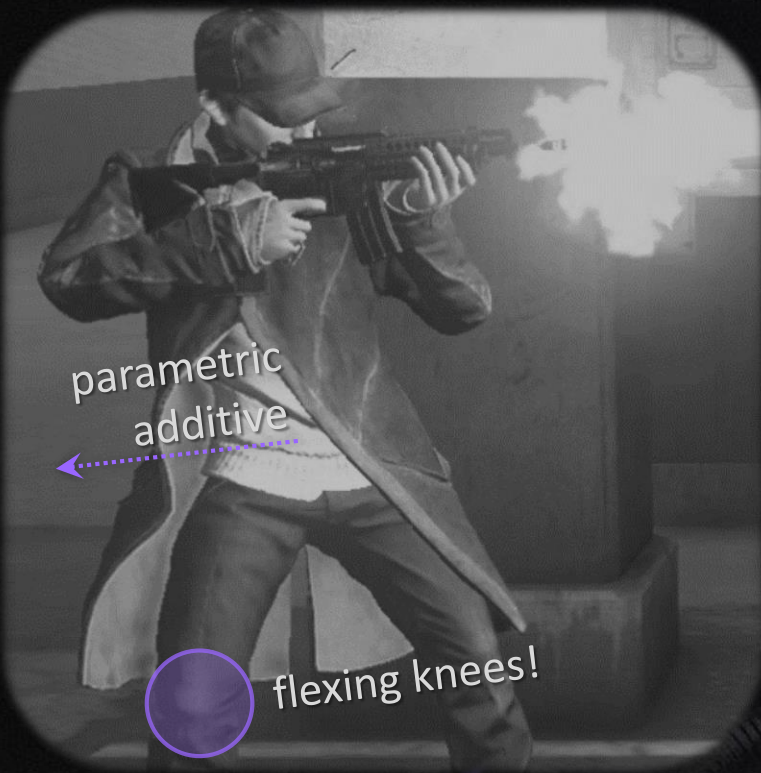
LAYERED AIMING ANGLE

2-BONE INVERSE KINEMATIC





## Multiple recoil layering for realism



WEAPON MECHANISM MOTION

WEAPON SKELETON MOTION

FULL BODY MOTION ADDITIVE

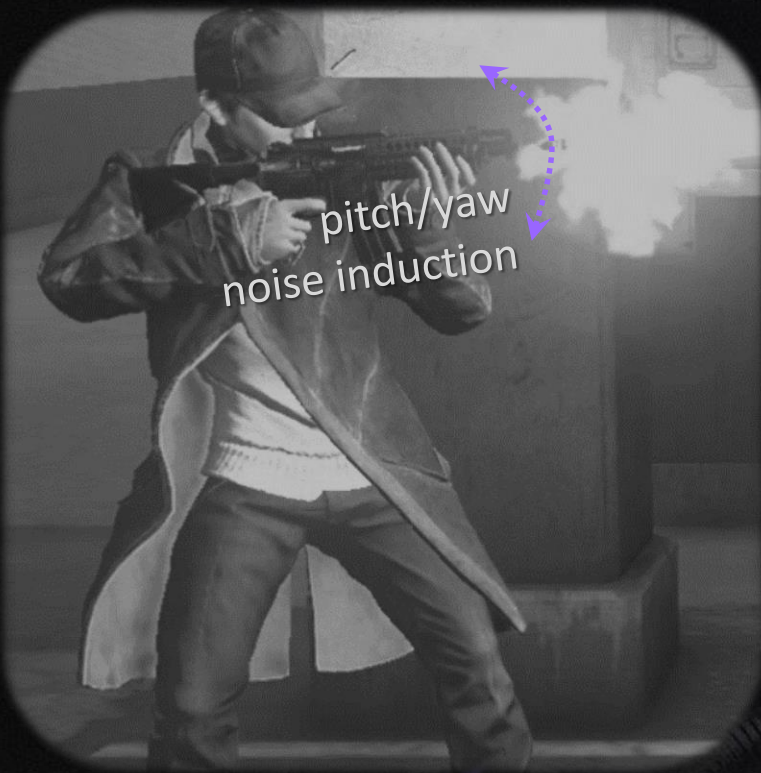
CAMERA GAMEPLAY MOTION

LAYERED AIMING ANGLE

2-BONE INVERSE KINEMATIC



# Multiple recoil layering for realism



WEAPON MECHANISM MOTION

WEAPON SKELETON MOTION

FULL BODY MOTION ADDITIVE

CAMERA GAMEPLAY MOTION

LAYERED AIMING ANGLE

2-BONE INVERSE KINEMATIC





## Multiple recoil layering for realism



WEAPON MECHANISM MOTION

WEAPON SKELETON MOTION

FULL BODY MOTION ADDITIVE

CAMERA GAMEPLAY MOTION

LAYERED AIMING ANGLE

2-BONE INVERSE KINEMATIC



15

Kinesic

Recoil layered system

VIDEO EXAMPLE

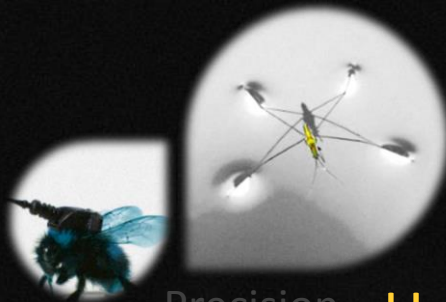


UBISOFT



15

Spatial Awareness Precision Usability Contextual  
Connectivity Fluidity Realism Seamless  
**Experience**  
Diversity Grounding Consciousness Body Awareness



UBISOFT

Simon Clavet Jack Potter Thomas Texier

Julien Bolduc Vincent De Perrot

Mathieu Huet

Éric Baillargeon Dan Brown Michel Cadieux  
Frederick Martel-Lupien

Félix Duchesneau Gilles Monteil

Colin Graham Frederick Zimmer

Benjamin Goyette Frederick Taylor

Louis-Vincent Fortin Simon Séguin Maxime Mercier

Vincent Chartrand Mattias Gruvman



*“ All fine architectural values are human values, else not valuable ”*

- Frank Lloyd Wright





15

# Questions ?

More ? Meet me @ the Ubisoft lounge!

**Thursday**

12.30PM @ 1.30PM

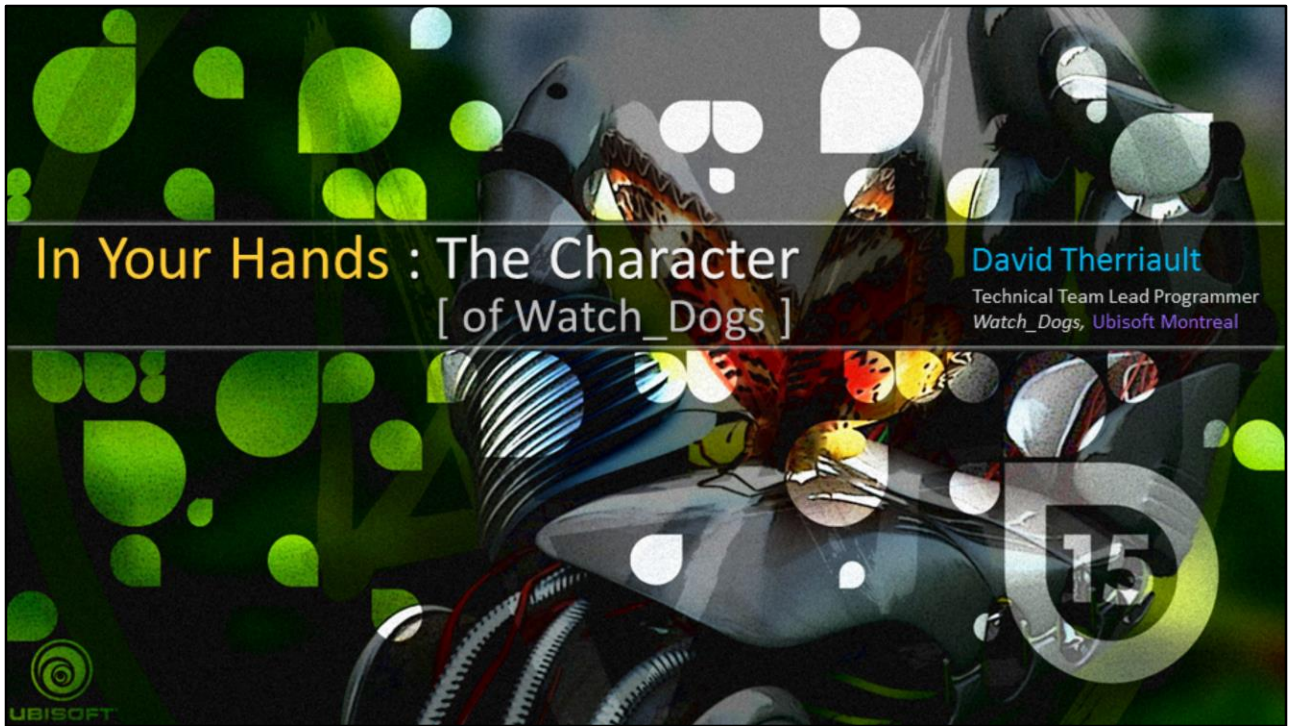
*West Hall, 3<sup>rd</sup> Floor*



UBISOFT

You can find the presentation's slides with some notes below... Cheers! 😊





# In Your Hands : The Character [ of Watch\_Dogs ]

David Therriault

Technical Team Lead Programmer  
Watch\_Dogs, Ubisoft Montreal

15





We are not here only for going to GDC parties...

We are here for **sharing** experience...

But also, most of all to **enhance** players' **experience** in the end!





The player's experience can be built by multiple things...  
But one of the most important is the **immersion quality factor**!



The player's experience can be built by multiple things...  
But one of the most important is the **immersion quality factor!**

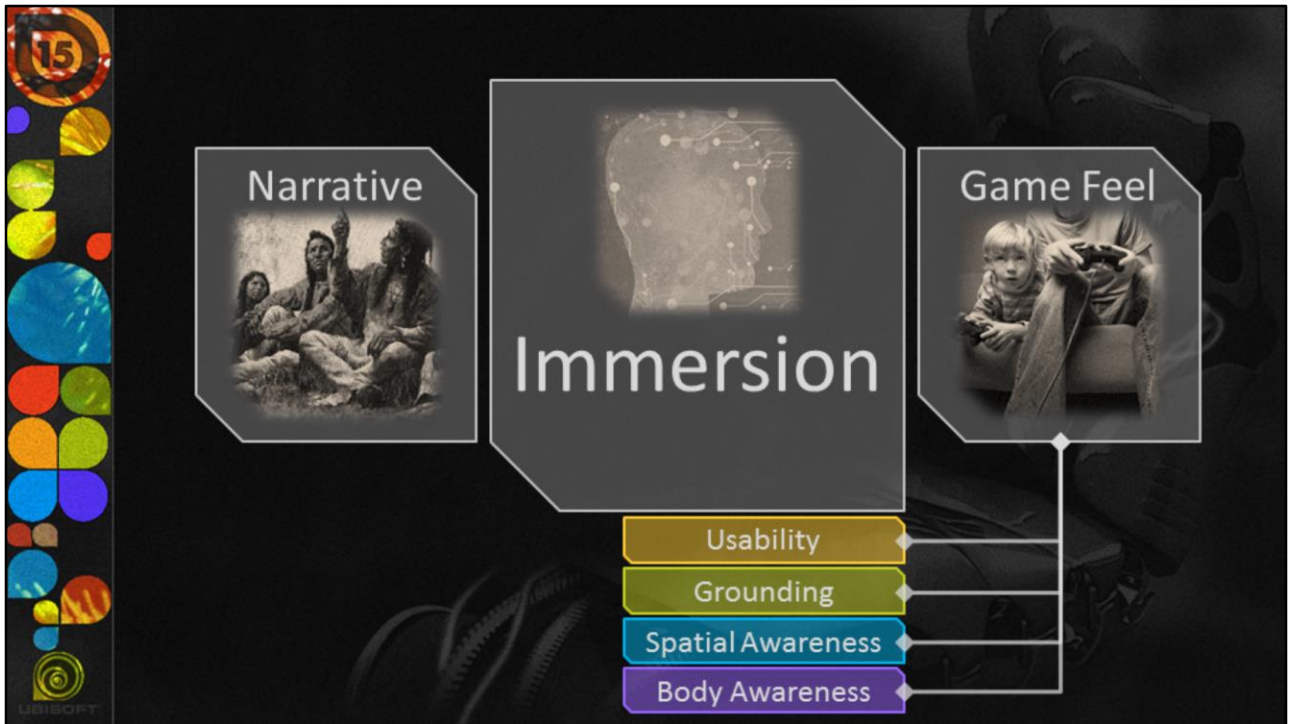




And immersion can be defined by two high level topics...

The **narrative aspects** of the game, which are not the focus of the presentation today

And the **game feel** of the game, which is going to be the hot topic of this talk!

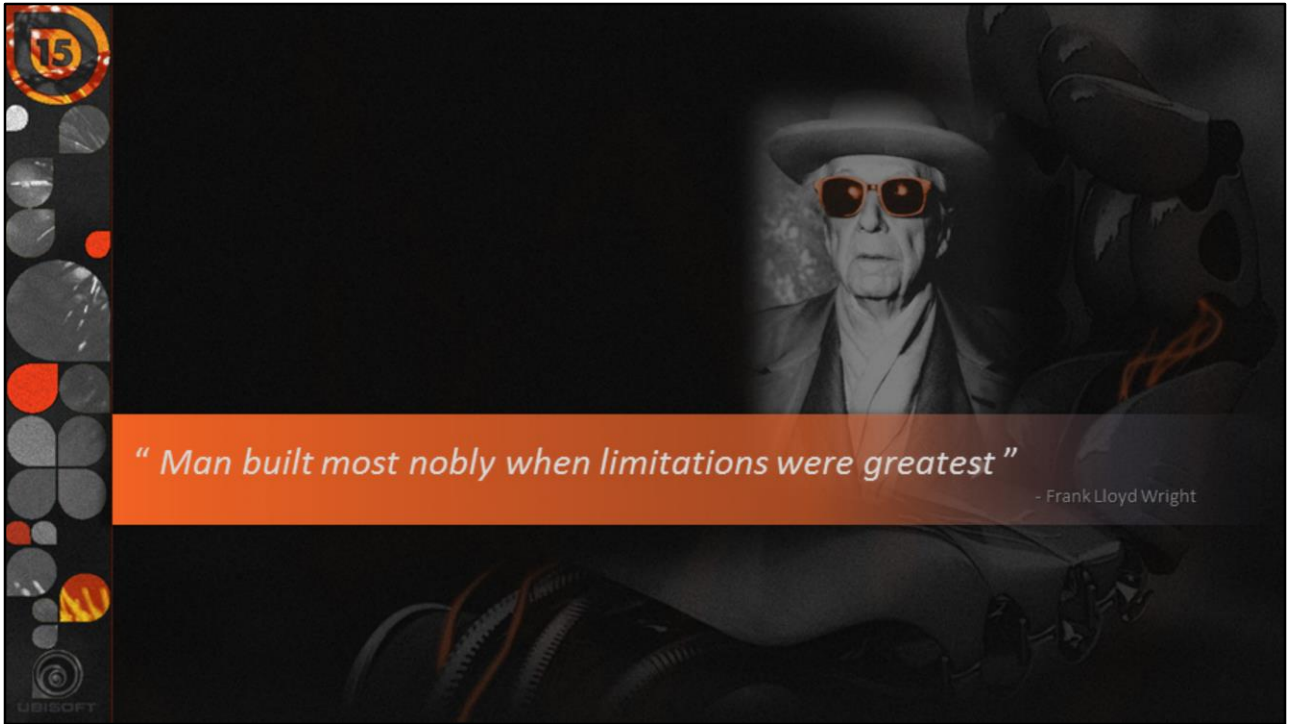


And the game feel can be divided into four big **categories** :

- **Usability** : for the **fluidity** and the **precision** of things
- **Grounding** : for the **contextual** and **seamless** activities
- **Spatial Awareness** : for the **consciousness** and the **diversity** of the environment
- **Body Awareness** : for the **realism** and the **connectivity** of the character itself

**Remember** these four categories because everything concerning the Watch\_Dogs state machine we will talk about today will be linked to these!

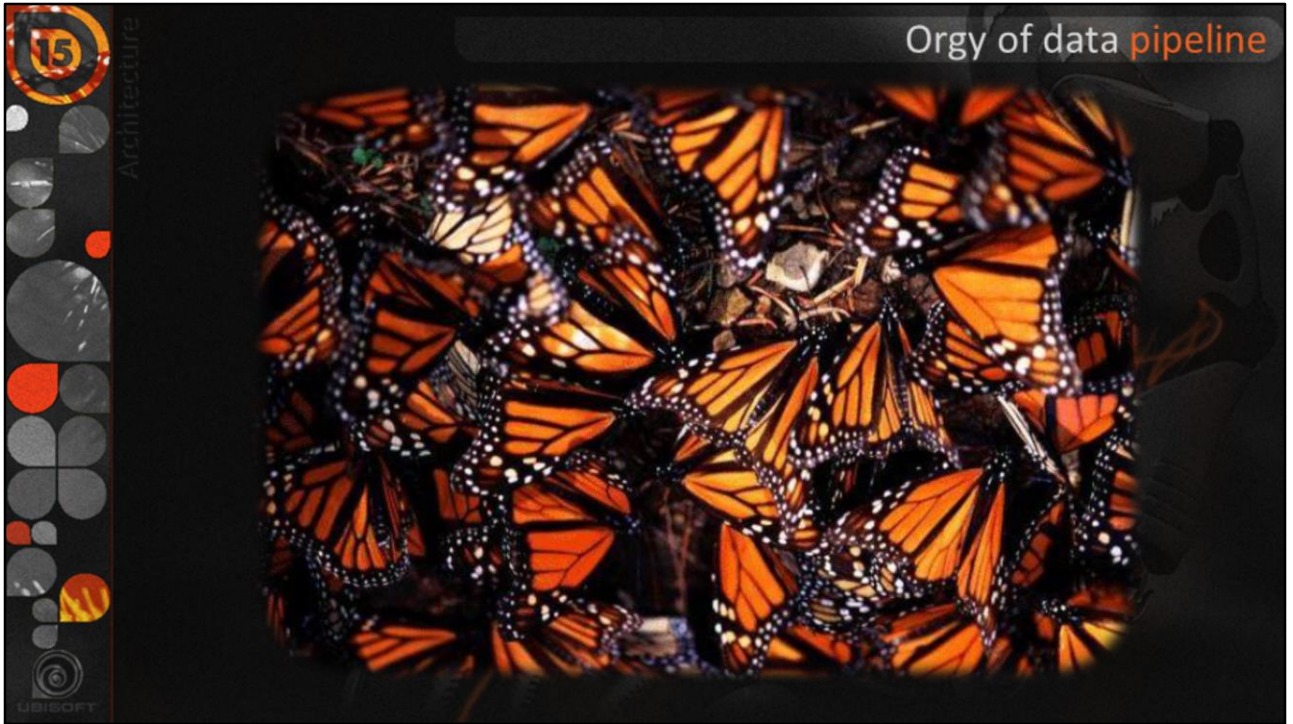




*" Man built most nobly when limitations were greatest "*

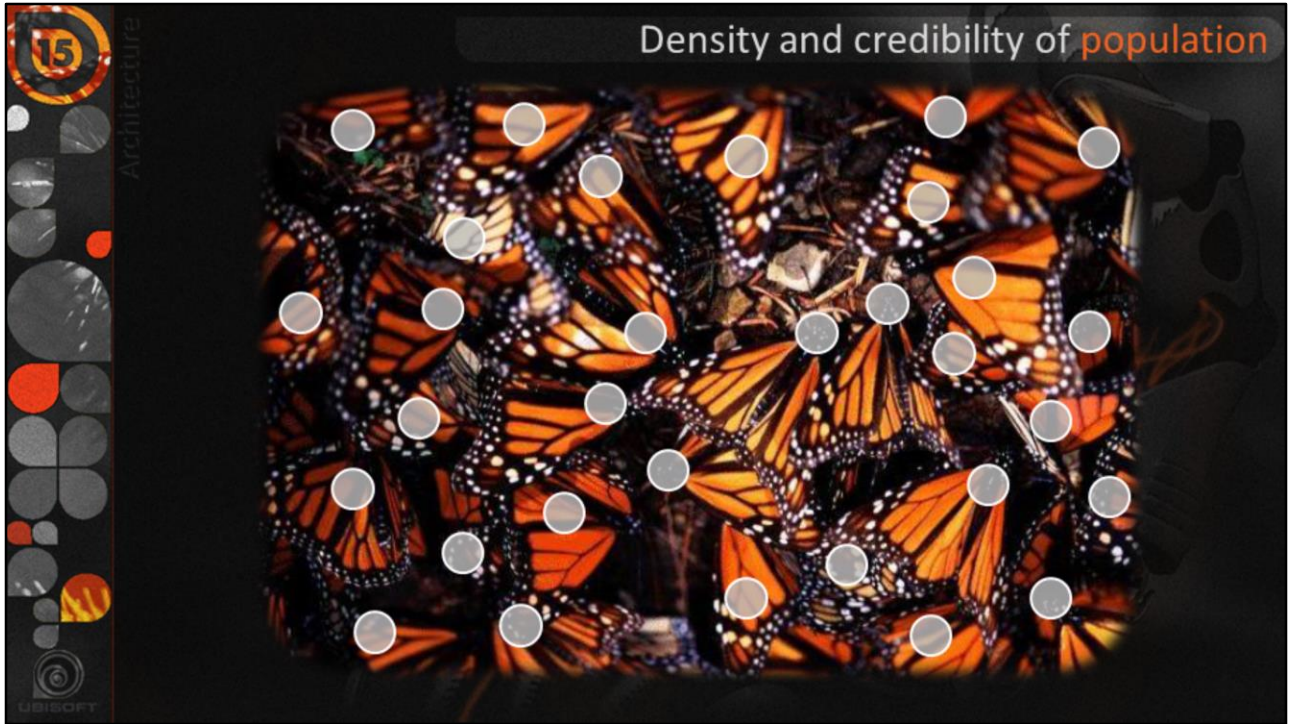
- Frank Lloyd Wright

Before diving into the architectural details of Watch\_Dogs state machine let's keep in mind this quote from a famous architect...



Our first limitation is that open-world games these days need to handle an orgy of data of all sorts!





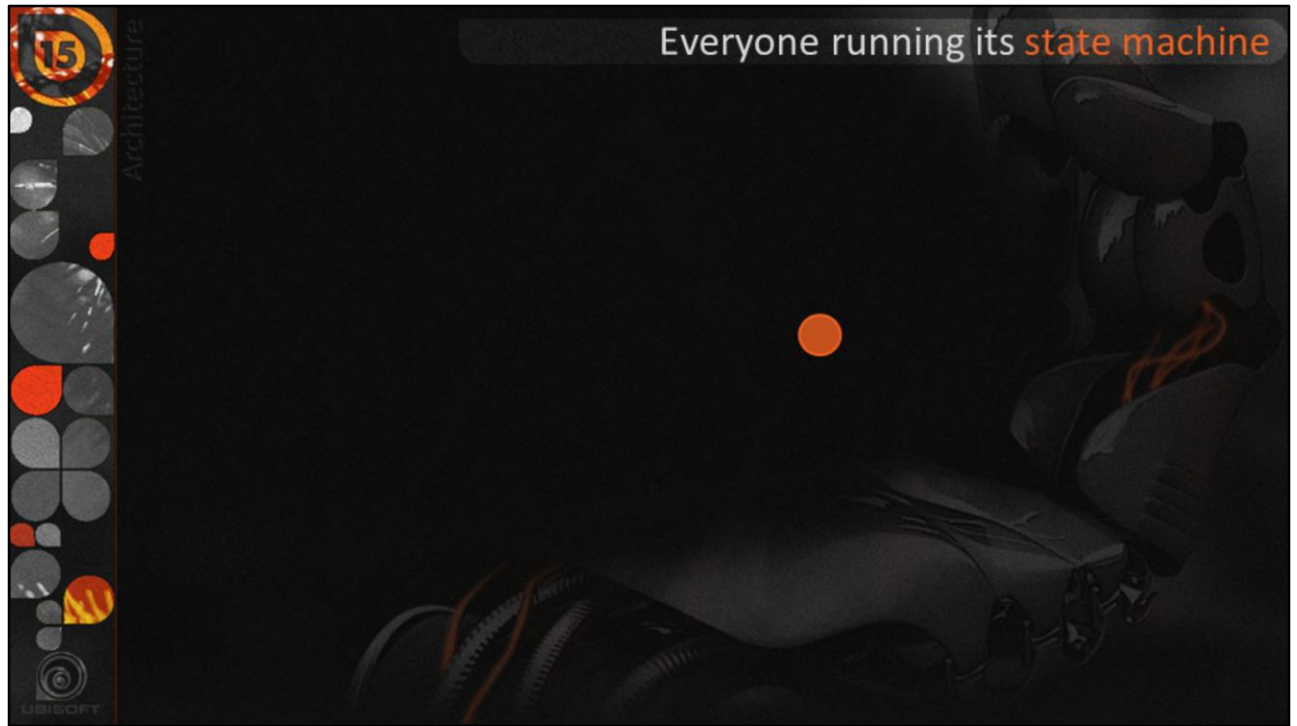
Our second limitation is that open-world games these days need a good density of population to boost immersion quality!



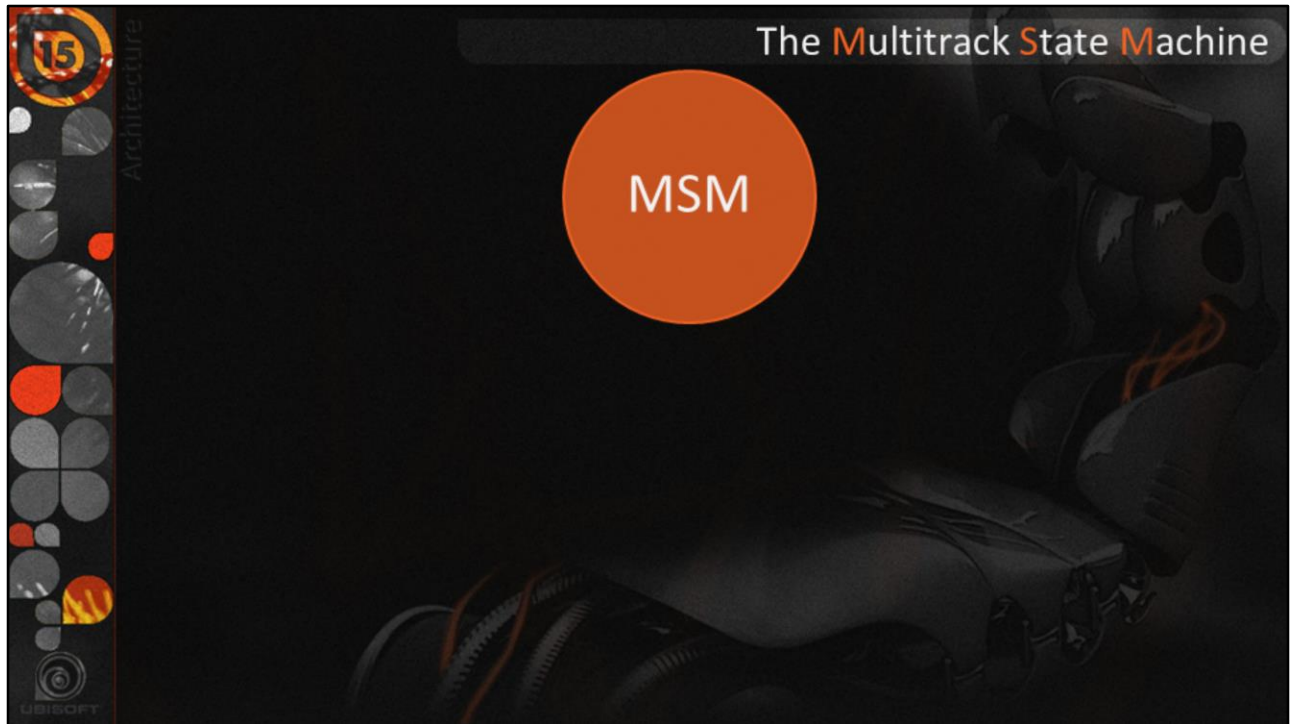
Our third limitation is that open-world games these days need to be online to live the game with friends live!

So, we need a lot of characters doing diverse credible things in the simulated world coming from a local instance or from a remote instance in our peer to peer architecture...



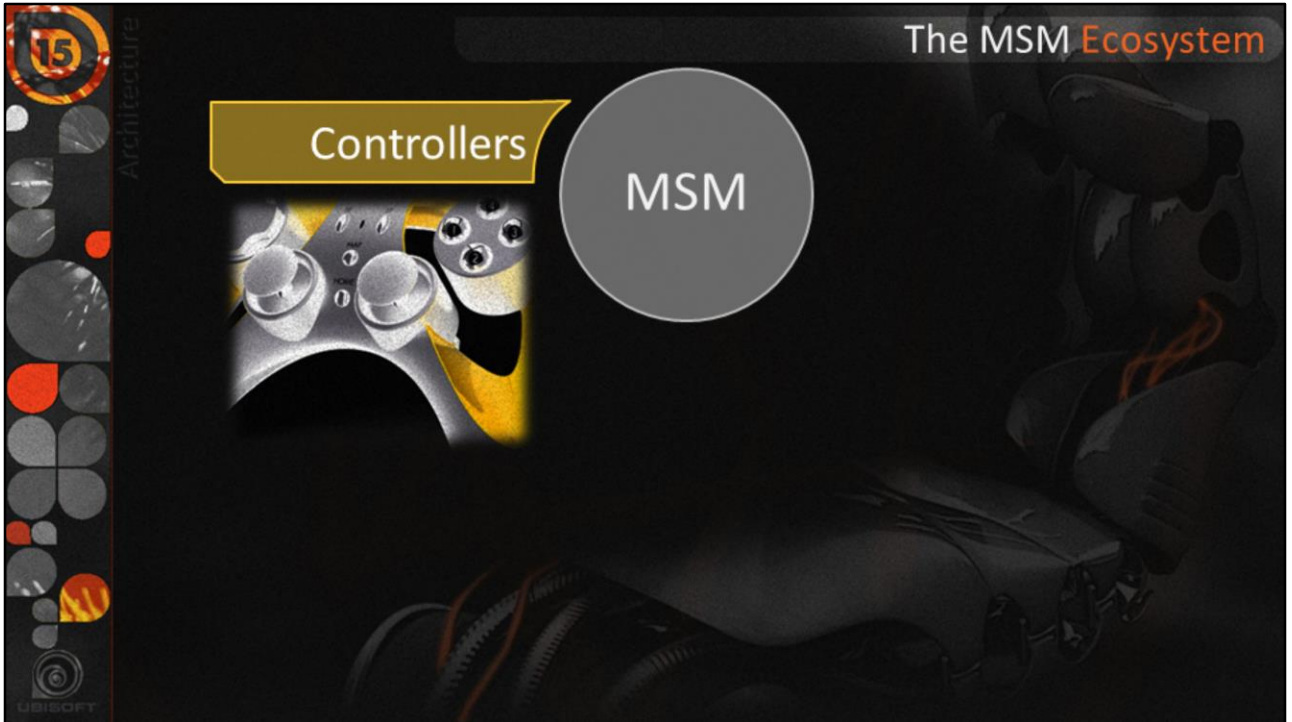


In the end...everyone is running its own state machine!



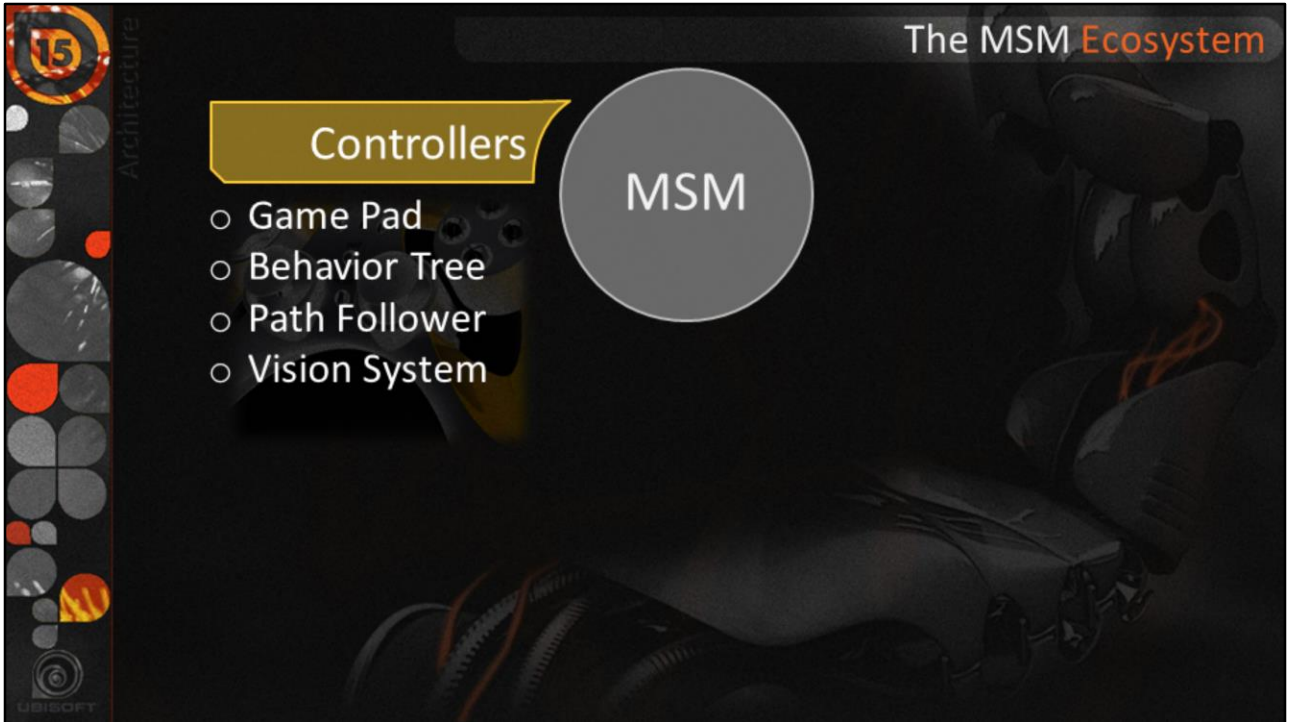
During Watch\_Dogs, we decided to build a custom **Multitrack State Machine**, which is going to be called the **MSM**.





Controllers (feeding **inputs** to the state machine at the **beginning of the frame**)

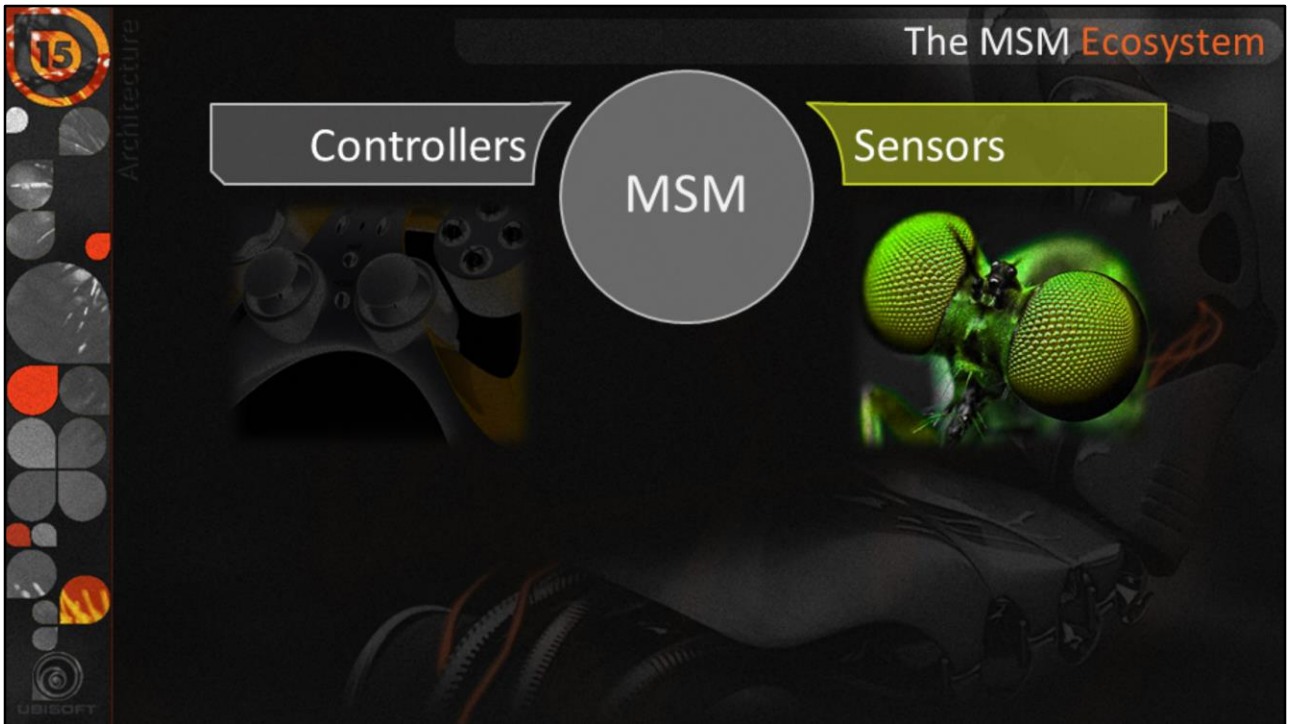
- Game Pad inputs
- Behavior Tree tasks
- Path Follower instructions
- Vision System computations



Controllers (feeding **inputs** to the state machine at the **beginning of the frame**)

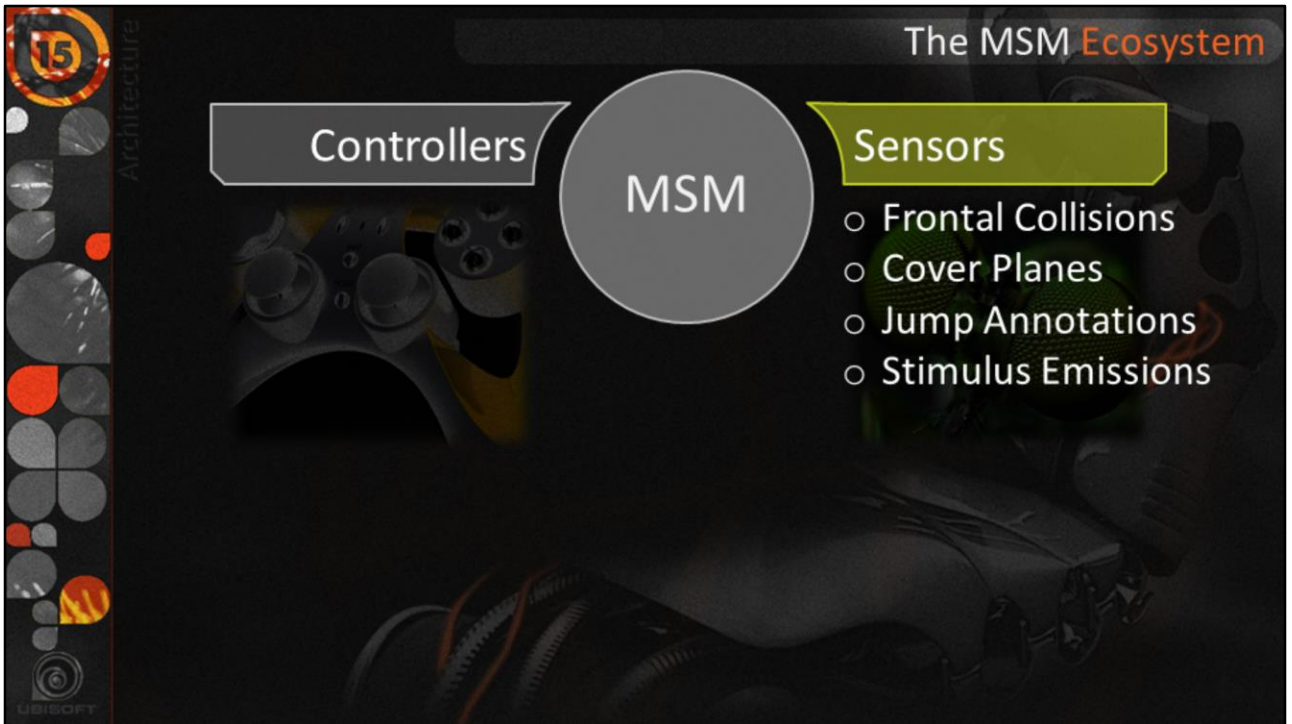
- Game Pad inputs
- Behavior Tree tasks
- Path Follower instructions
- Vision System computations





Sensors (feeding **inputs** to the state machine **asynchronously during the frame**)

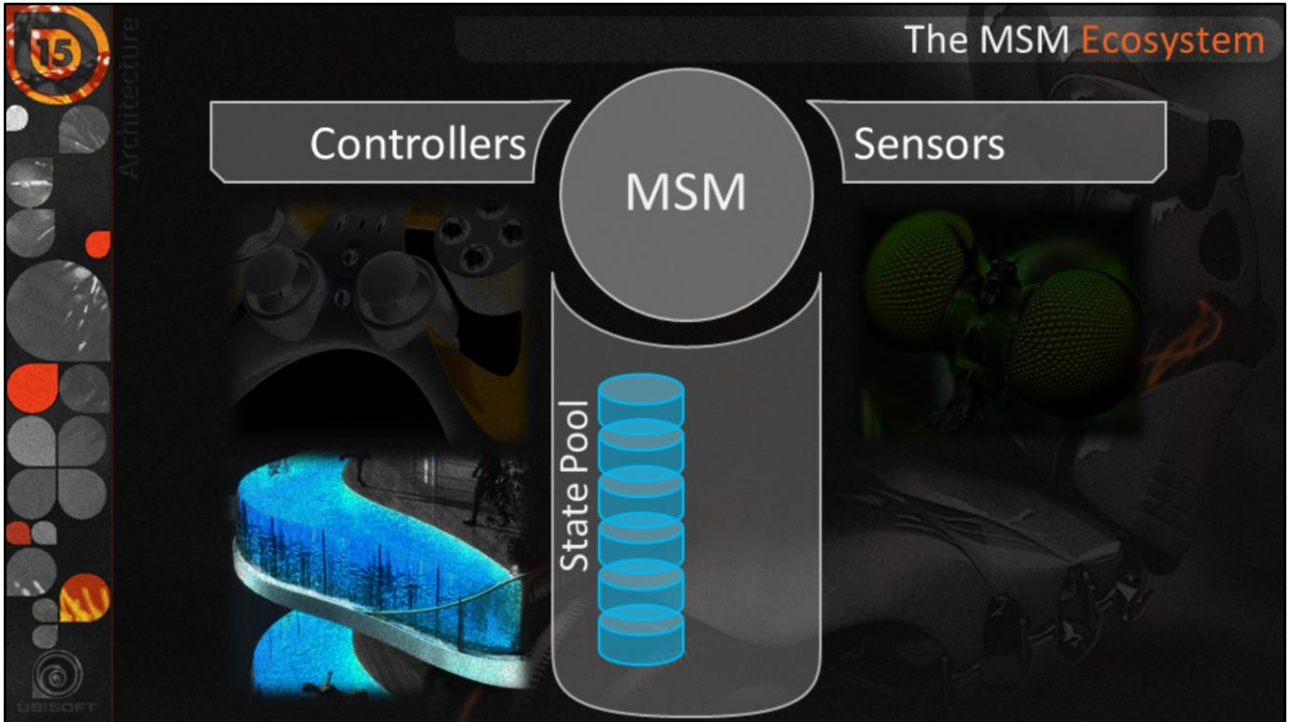
- Frontal Collisions reports
- Cover Planes targets
- Jump Annotations possibilities
- Stimulus Emissions events



Sensors (feeding **inputs** to the state machine **asynchronously during the frame**)

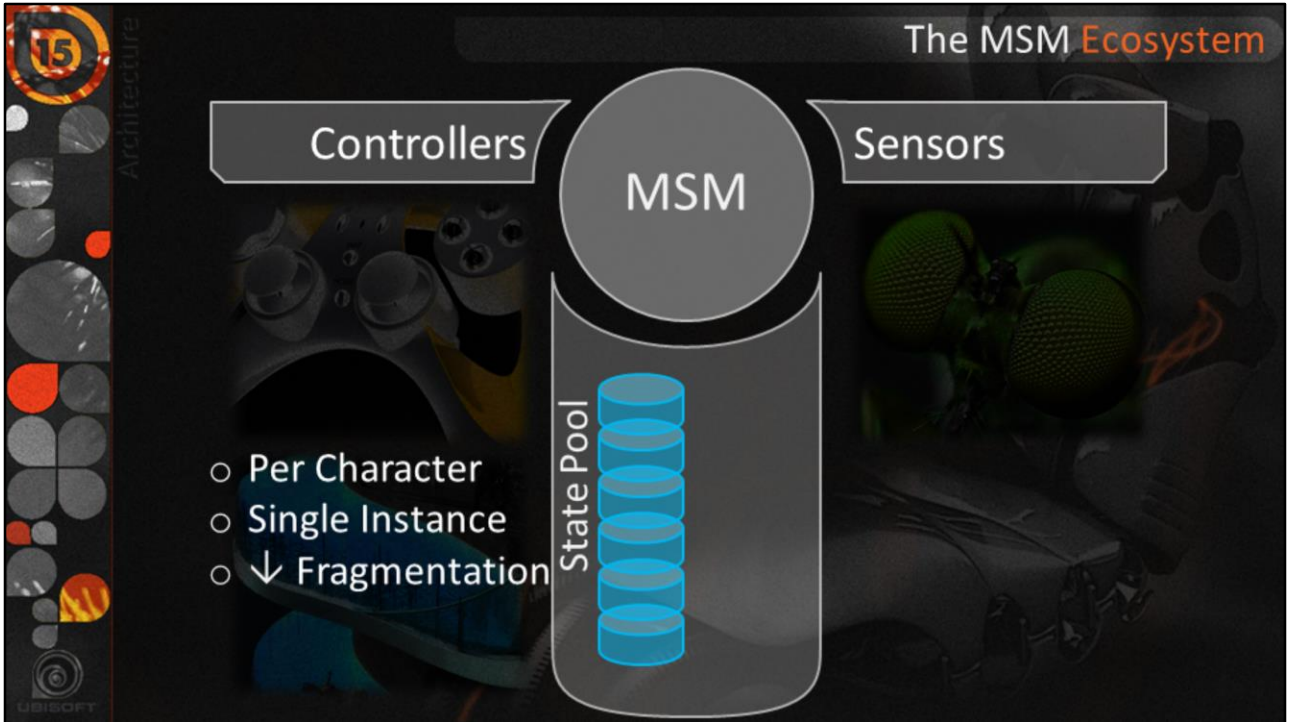
- Frontal Collisions reports
- Cover Planes targets
- Jump Annotations possibilities
- Stimulus Emissions events





State Pool (being the **definition** of the state machine)

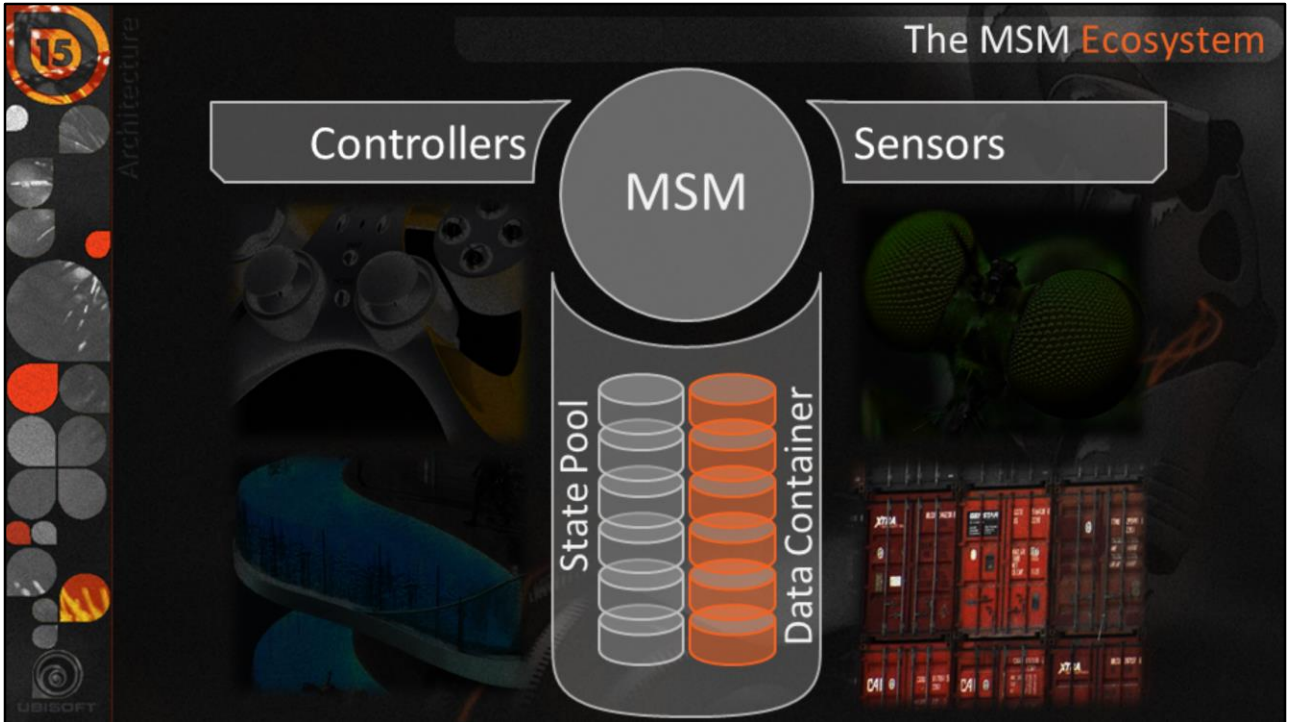
- One pool per character which is defining its state **domain**
- One and only one single instance of each state in the pool for **hypothesis & executions**
- Causing less memory fragmentation by keeping the **footprint** pretty stable once the character is initialized



State Pool (being the **definition** of the state machine)

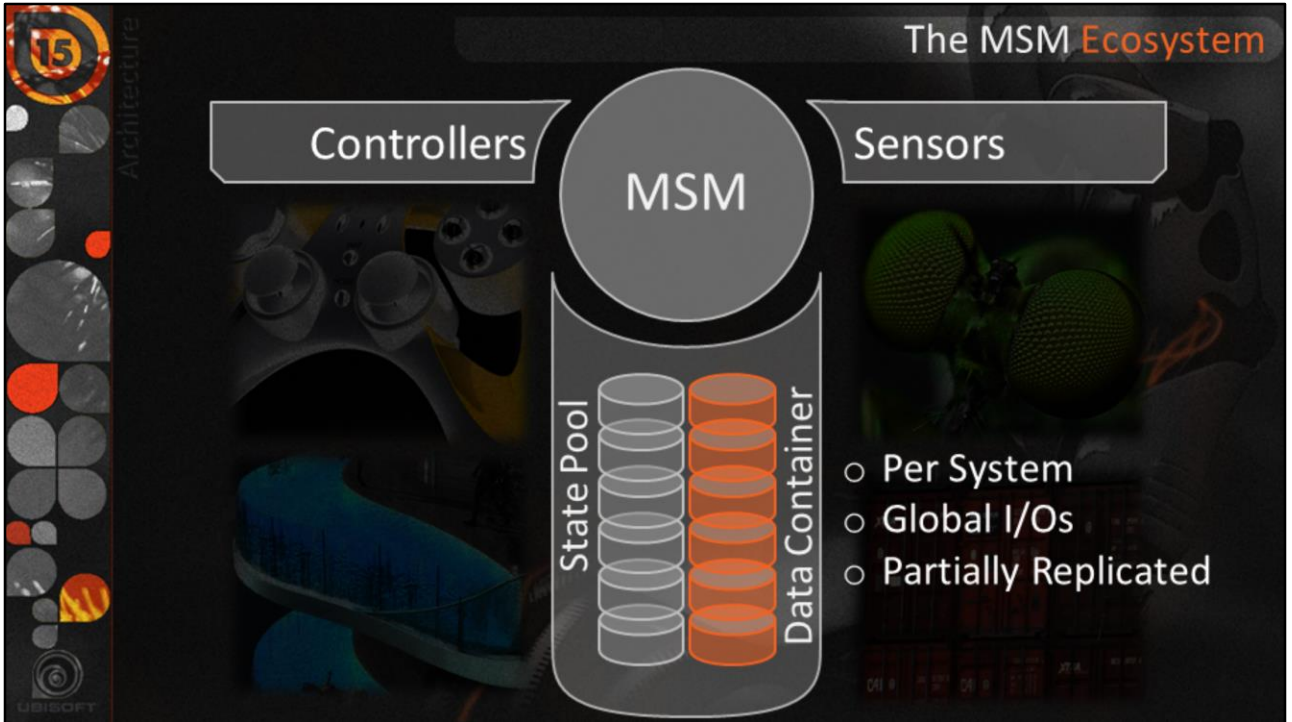
- One pool per character which is defining its state **domain**
- One and only one single instance of each state in the pool for **hypothesis & executions**
- Causing less memory fragmentation by keeping the **footprint** pretty stable once the character is initialized





Data Containers (being the input/output **storage** of the character **status**)

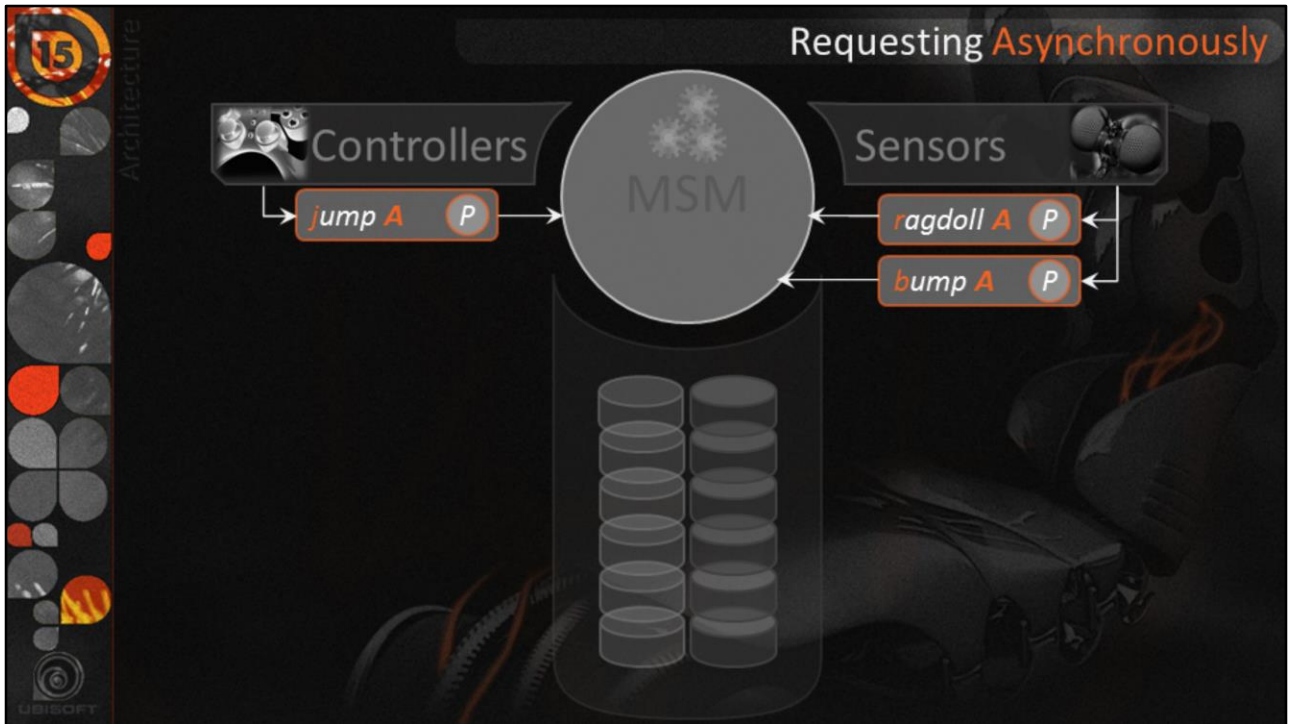
- One container per system **sharing** various information like speeds, positions, object handles, etc...
- Allowing systems like the state machine to read or write this status **information to run**
- Replicated partially because we want to **minimize the bandwidth** usage in between hosts & replicas



Data Containers (being the input/output **storage** of the character **status**)

- One container per system **sharing** various information like speeds, positions, object handles, etc...
- Allowing systems like the state machine to read or write this status **information to run**
- Replicated partially because we want to **minimize the bandwidth** usage in between hosts & replicas

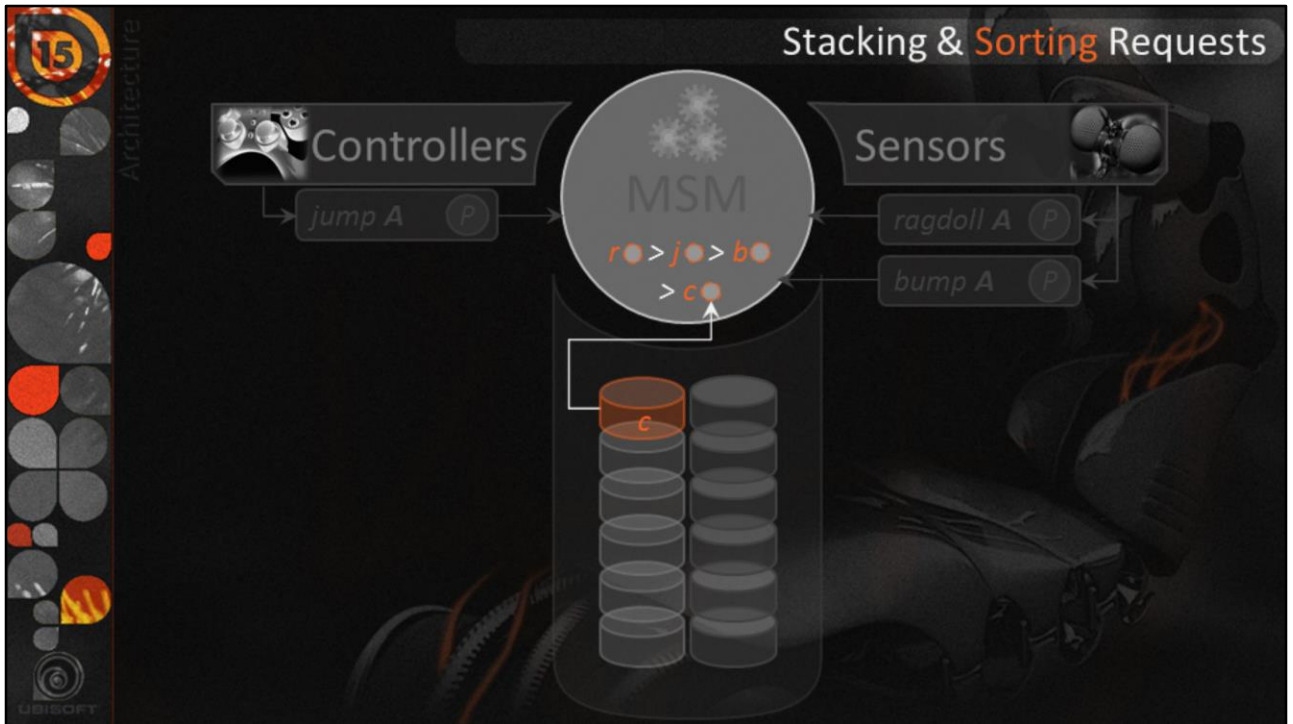




Now let's look at a simple example about how these elements **collaborate** with each other to make the MSM run

Controllers & Sensors are able to send **asynchronous** state requests

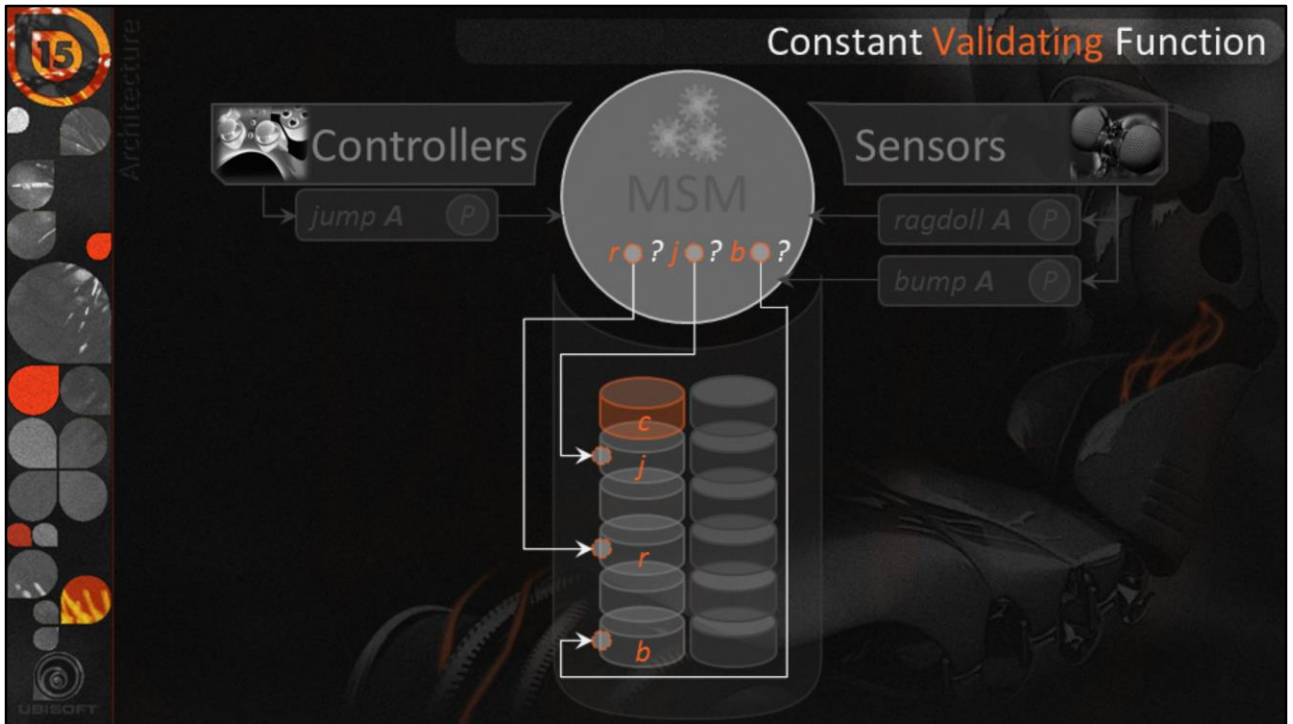
- State request object **built locally** and then sent by event to the MSM
- State request object made of the **state definition** which is a **lightweight** structure of the state
- State request object also made of the **start up parameters** filled by the requesting code



Once these state requests are **stacked** inside the MSM, they will be **consumed** at the beginning of the frame

- First pass of consumption is **priority sorting** using the state definition object within the request object
- Second pass of consumption is **priority comparison** with the current state running



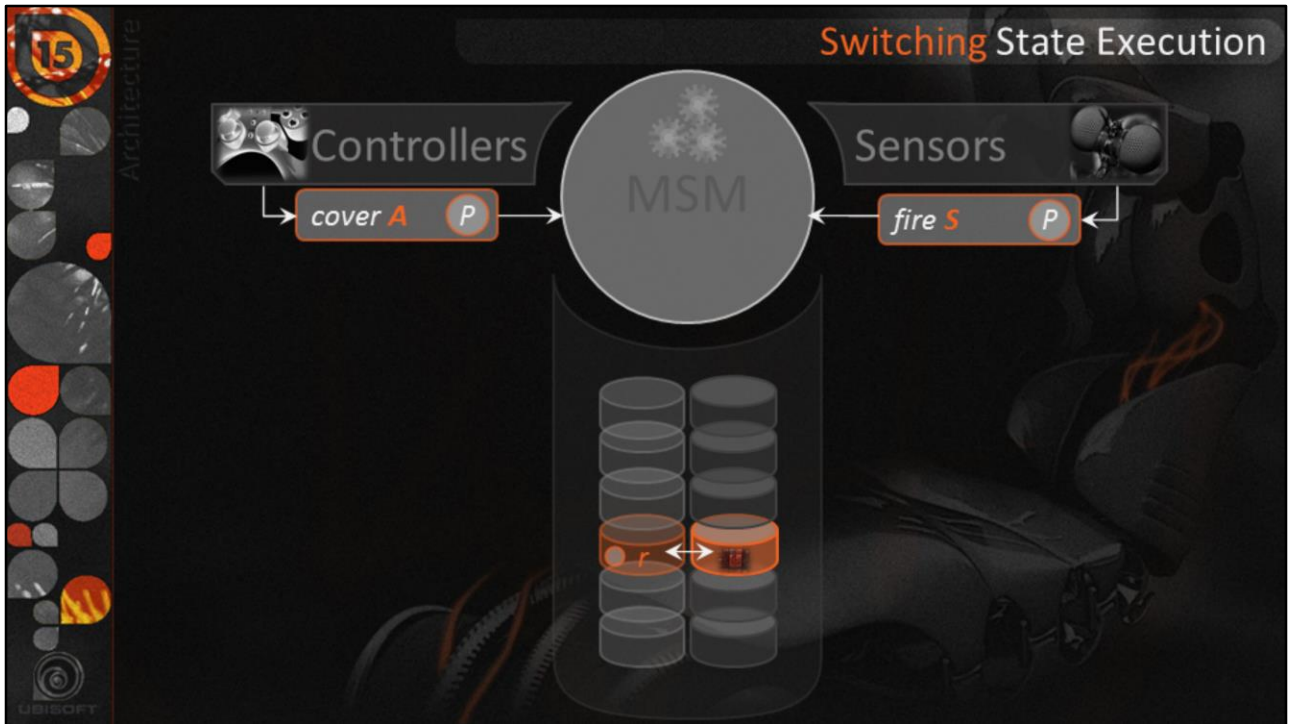


Once they are well sorted in between themselves, we proceed with their validation

- First step is extracting the **start up parameters** stored in the state request object
- Second step is pushing these start up parameters to the **corresponding state** in the pool

Every single instance of these states have a **const validating function** to evaluate the parameters even if the state is currently running

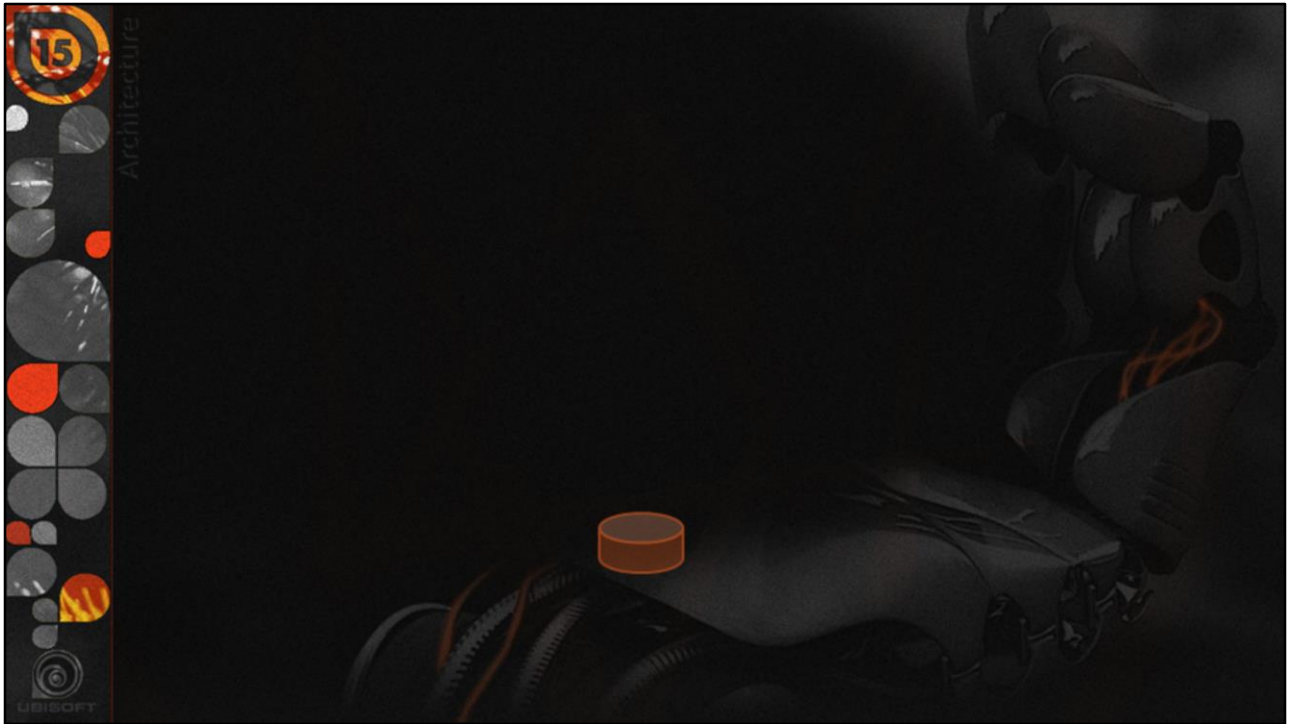
Because we **keep the same** start up parameters object from the beginning of the request, the state can **cache expensive computing** in it also



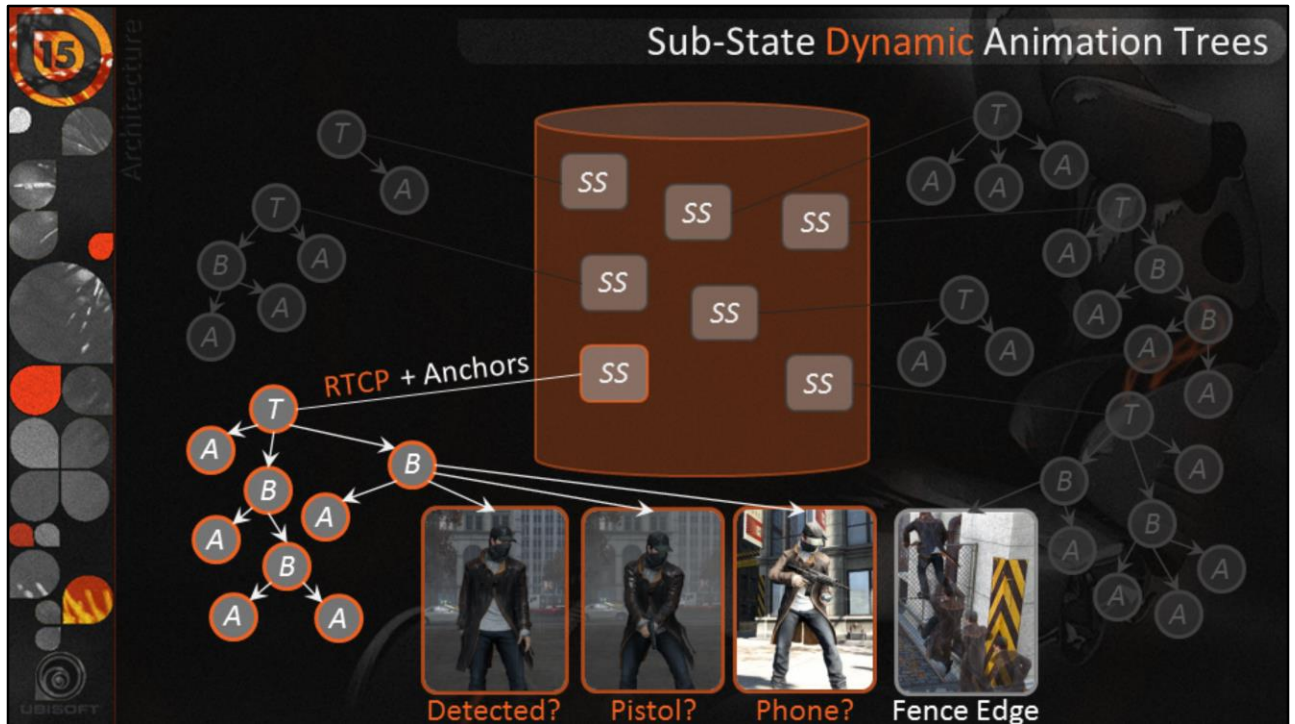
Once the **highest priority state passed** it is elected as the running state of the MSM

During its execution it will use the **necessary data containers** to read/write **useful information** to be able to run properly. And while it is being executed, the **ecosystem will continue** to request other states





Let's dig deeper into the internal state mechanics...



One particularity of this multi track state machine is that every state **doesn't have infinite depth** like a hierarchical state machine

Deliberately **forced only 1 layer of sub states** inside each state and **common services** in between states are contained **within the class hierarchy**

Every sub state can be or not **referencing a dynamic animation tree** from which we build the **resource dependencies** for each state pool

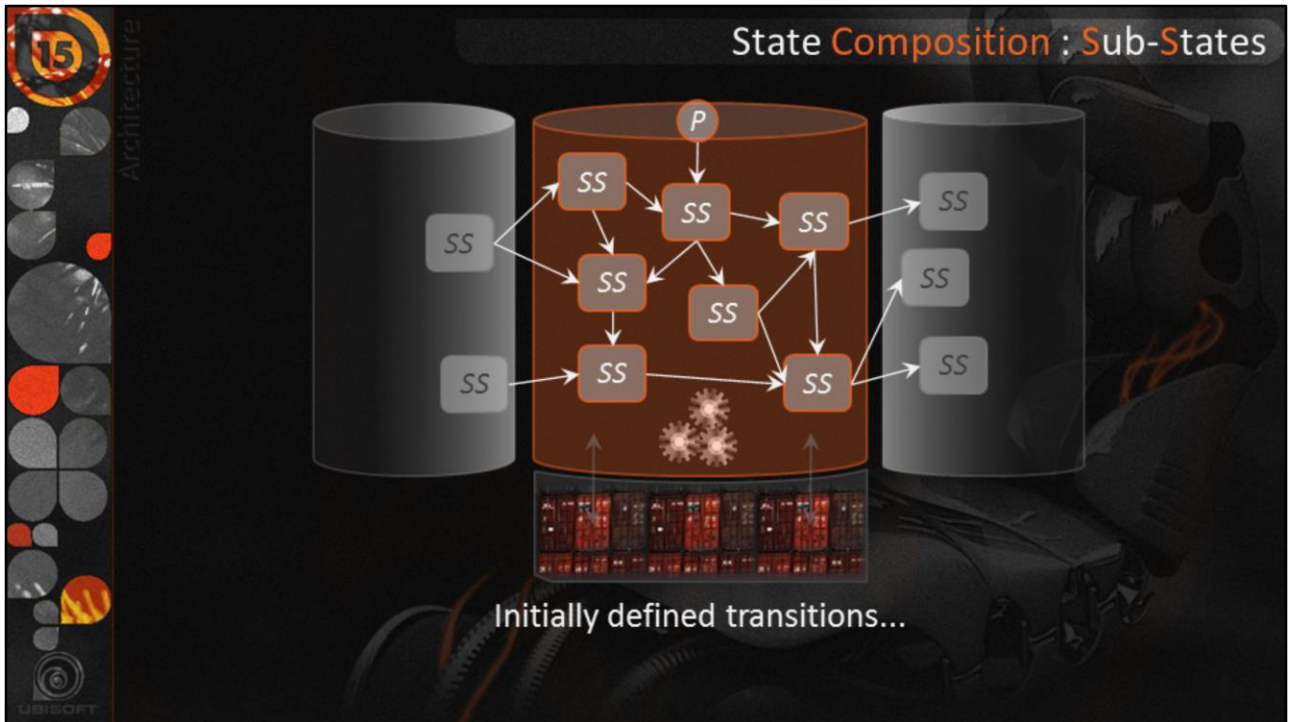
**Decision trees** are logical trees and **Blend trees** are leafs of them

Substates are responsible to push proper **real-time control parameter structure**

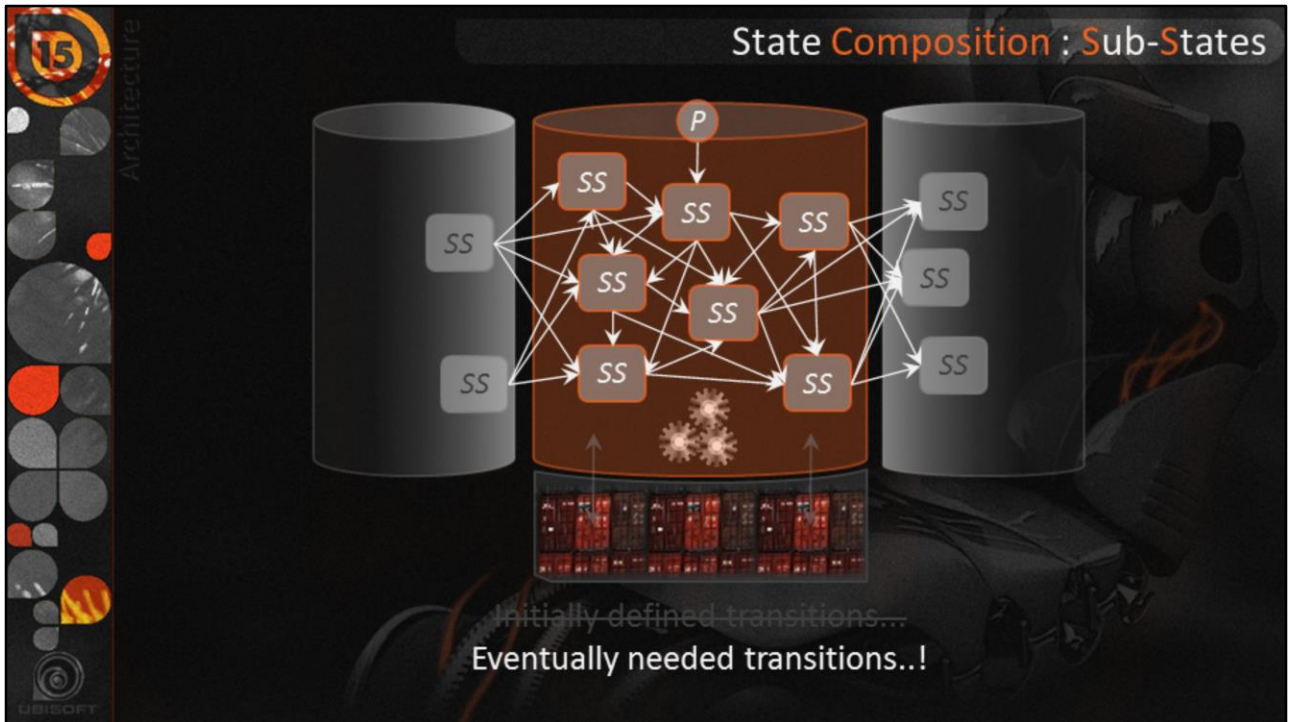
Decision trees and blend trees **evaluate the proper** animations to play based on this RTCP data structure

Substates are also responsible to push environmental information called **anchors**





So when the state starts, it **receives the start up parameters** to decide what to do...  
Then the good sub state will run with precise information from them  
At first we had a graphical tool to express **states & substates possible transitions**



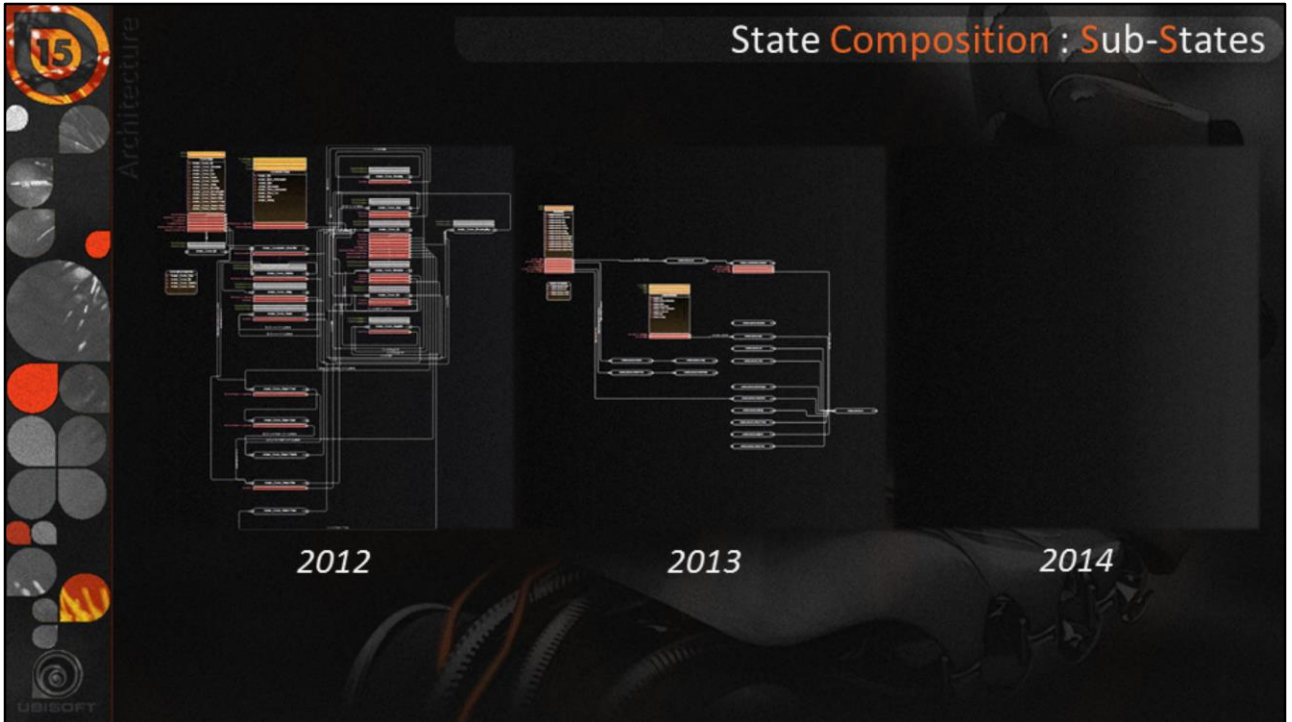
But in the end, it turned out to be a total mess to understand because everything was linked to everything...

Mainly because we want to be **easily interruptible** but also because when you replicate a state you might **miss network packets**!





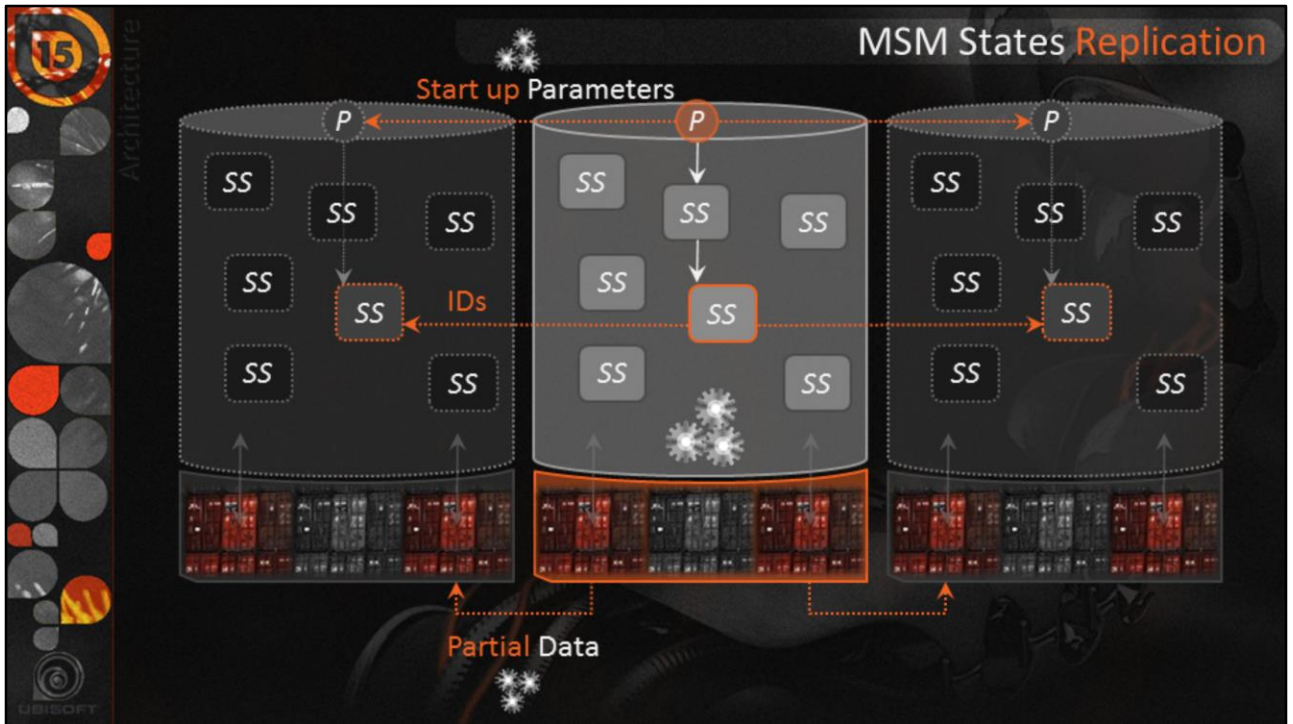
## State Composition : Sub-States



As a concrete example, here is the cover state graph with all its substates in 2012...  
And then we started removing useless internal links during 2013...  
To finally completely remove this graphical tool from our state machine pipeline in 2014...

In the end, good **programmers are lazy** by definition...They don't want to manage a graphical state machine if they have to manage the code state machine on the other side!





So we removed all these links because we needed to network efficiently the MSM.

There are 3 main things we need to replicate to achieve a “kinda” deterministic state machine :

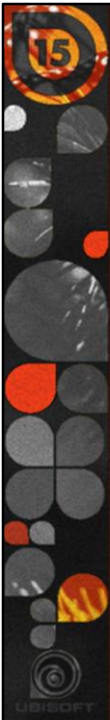
- **Start up parameters to synchronize the state activation**
- **State & Substate unique IDs** to drive the internal flow
- **Data containers partial data** replication system

Good **programmers are lazy!** So we created a way to abstract the replication process the most possible...

By default, the whole start up parameters will be **automatically packed** and sent to replica's machines

And in parallel, the data containers will **compress and send tagged members** only to the replica's machines

This is done by a special macro system within the C++ to auto generate the accurate members for the network object model. A float is a float for the gameplay engine layer but it is a **compressed resolution** float for the network layer



```

class CPlayerCoveredMotionStateDef : public COnFootMotionStateDef
{
    DECLARE_STATE_DEF_PARAMS(CPlayerCoveredMotionStateParams);

    DECLARE_STATE_DEF_BEGIN(CPlayerCoveredMotionStateDef, COnFootMotionStateDef, eStatePriority_High)
        DECLARE_STATE_DEF_SUBSTATE(Approach)
        DECLARE_STATE_DEF_SUBSTATE(Enter)
        DECLARE_STATE_DEF_SUBSTATE(Idle)
        DECLARE_STATE_DEF_SUBSTATE(Start)
        DECLARE_STATE_DEF_SUBSTATE(Move)
        DECLARE_STATE_DEF_SUBSTATE(Stop)
        DECLARE_STATE_DEF_SUBSTATE(TurnAnticipation)
        DECLARE_STATE_DEF_SUBSTATE(Turn)
        DECLARE_STATE_DEF_SUBSTATE(SpeedTurnTower)
    DECLARE_STATE_DEF_END()
}

class CPlayerCoveredMotionStateParams : public CBaseStateParams
{
    DECLARE_STATE_PARAMS(CPlayerCoveredMotionStateParams, CBaseStateParams)

    DECLARE_NETDATA_BEGIN(CPlayerCoveredMotionStateParams, CBaseStateParams::NetData)
        DECLARE_NETDATA_MEMBER(m_coverEntryAttachment, CCoverAttachment, CNetDataCoverAttachment)
        DECLARE_NETDATA_MEMBER(m_randomValue, ndFloat, storm::DataFloat<1, storm::NetDataFloatEpsilon_0_0>)
        DECLARE_NETDATA_MEMBER(m_skipEntry, ndBool, storm::DataBool)
    DECLARE_NETDATA_END()
}

CStateRequest<CPlayerCoveredMotionStateDef> coverRequest;

coverRequest.GetParams().SetCoverEntryAttachment(coverAttachment);
coverRequest.GetParams().SetRandomValue(ndRandFloat());

coverRequest.Send();

```

Here is a **small glimpse** at how things work **automatically** in the code for the motion cover state for example :

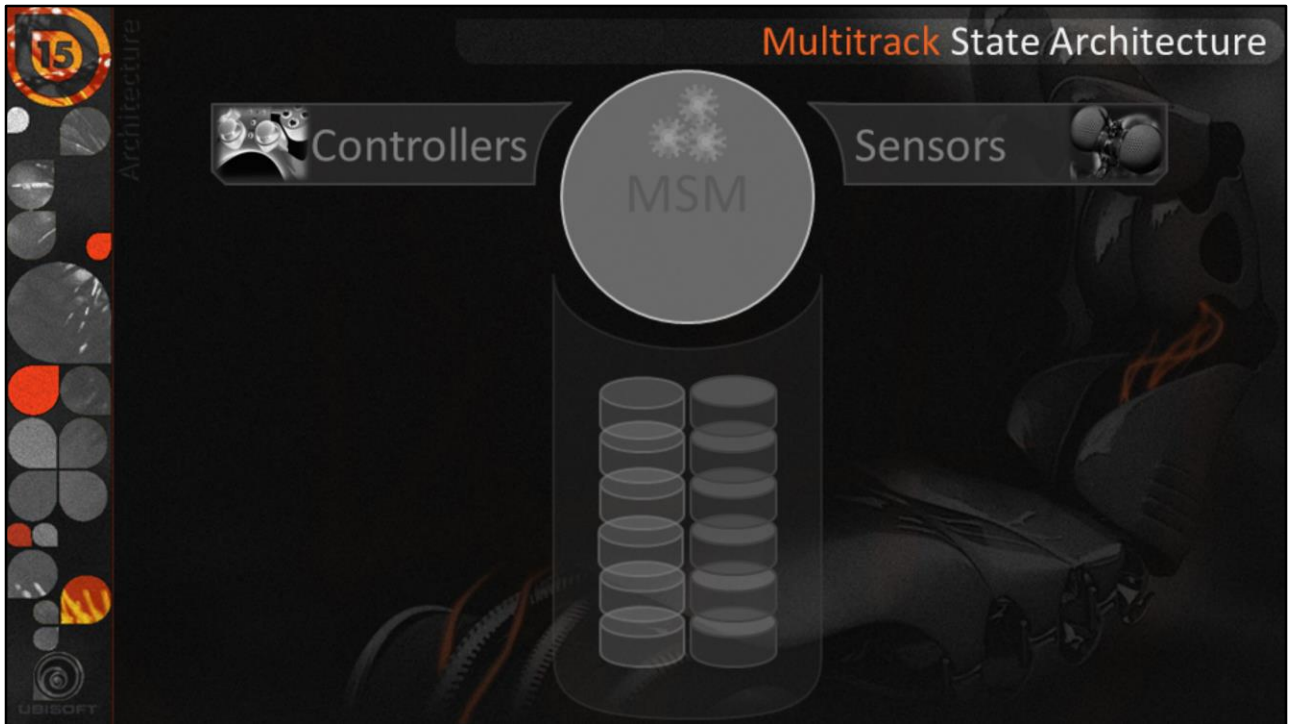
- At the top, straight forward **sub state definition** with one liners
- In the middle, straight forward **start up parameters networking** with one liners
- At the bottom, straight forward **state request generation & emission** with one liners





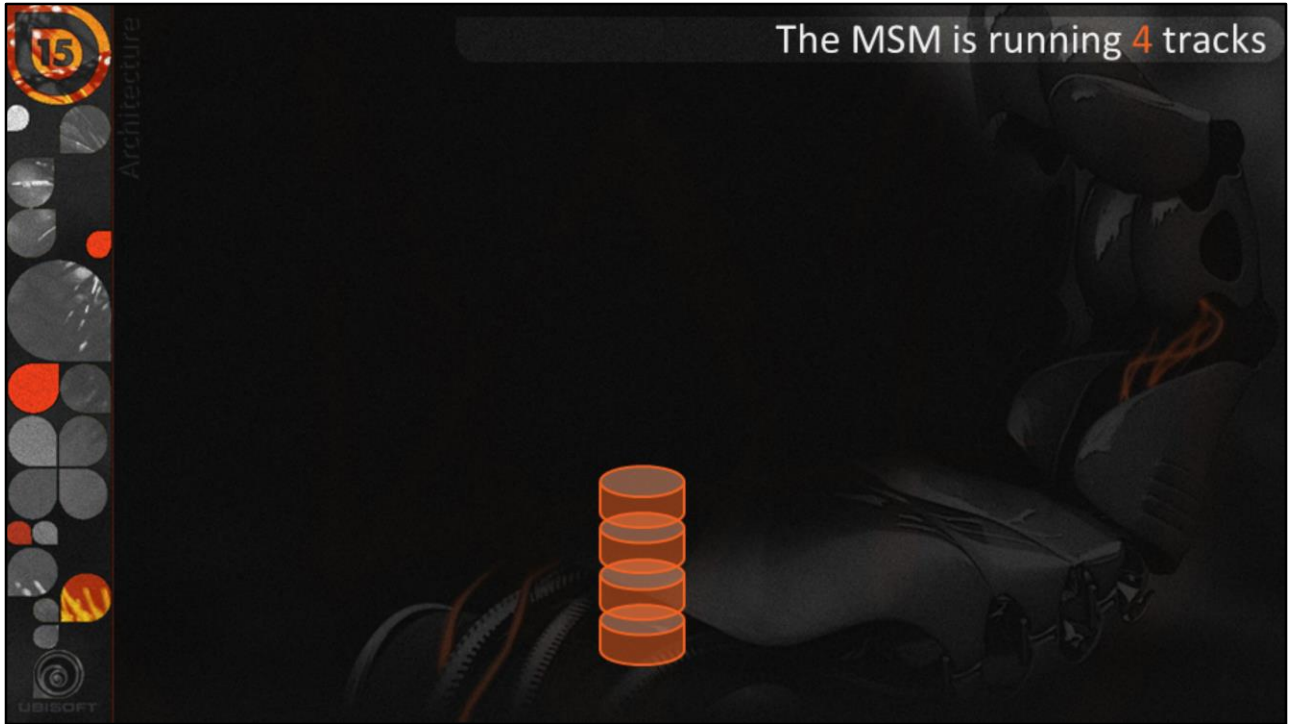
And then...you'll need good tools to debug your animations playing...  
The best one we used on Watch\_Dogs was called the Ghost feature  
Creating a **local replica** of the player using the local machine **network loopback**

That way you can easily simulate **random lags**, **packet losses** and see both entities in the same screen!

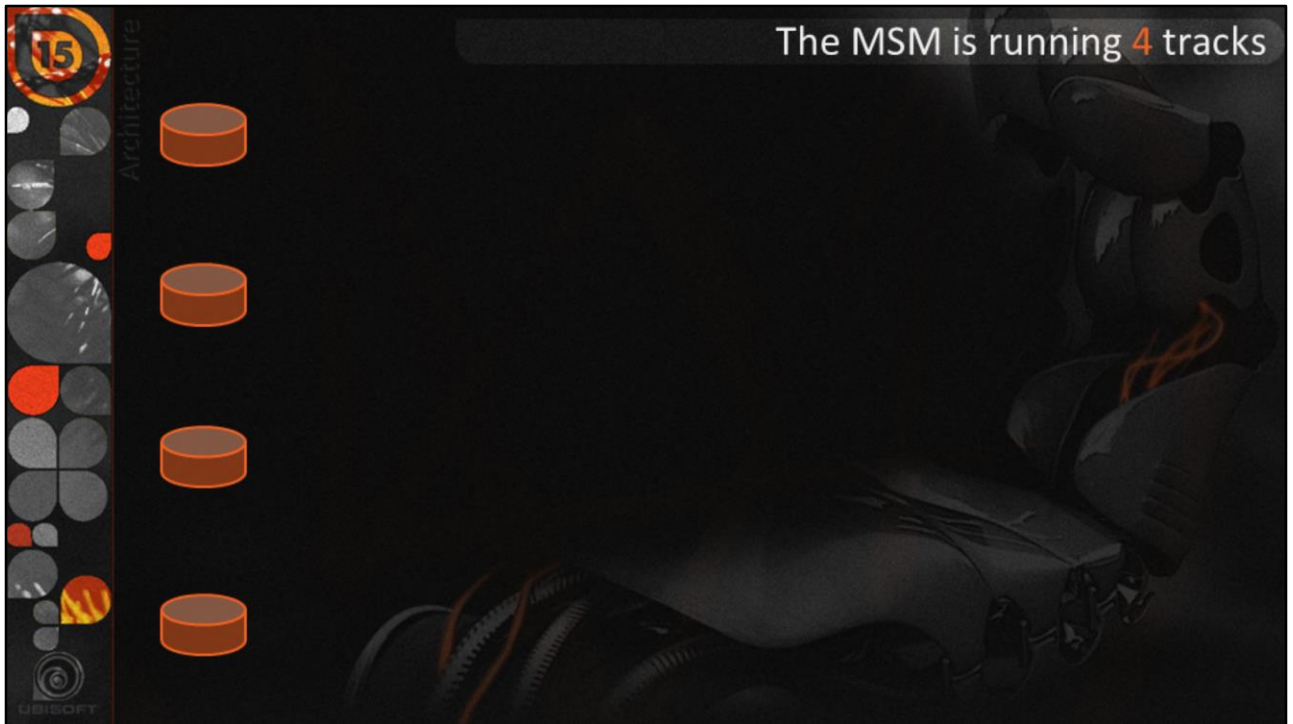


Since the beginning we are talking about a multi track state machine...But where and what are these tracks !?





Another limitation we forced was to cap the number of tracks to **4** and to dedicate each one of them to a **responsibility**



Another limitation we forced was to cap the number of tracks to **4** and to dedicate each one of them to a **responsibility**





The first one is called **Motion**, move the character from **A to B**

The second one is called **Action**, do things requiring the **whole body**

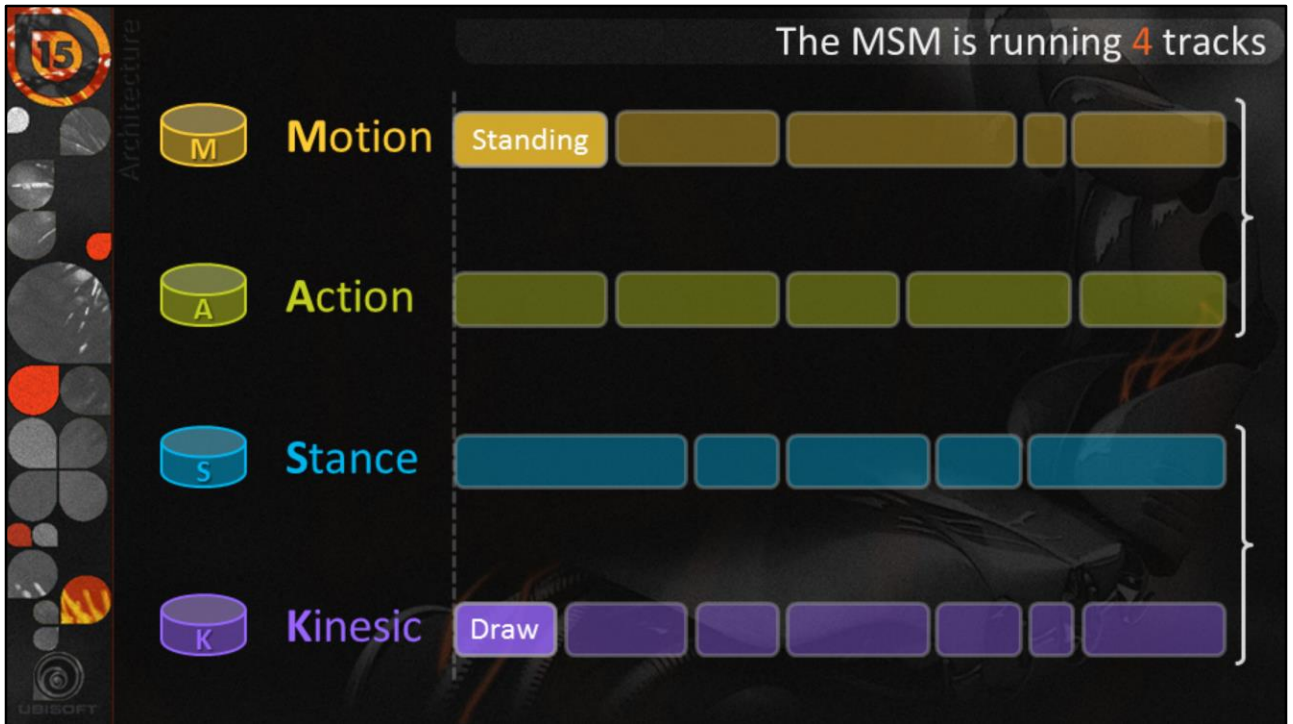
**Motion & Actions** are sharing the full body layer within the animation engine, which means they play one dynamic tree at the time together

The third one is called **Stance**, convey the **context** of the character

The last one is called **Kinesic**, represent hand **manipulations**

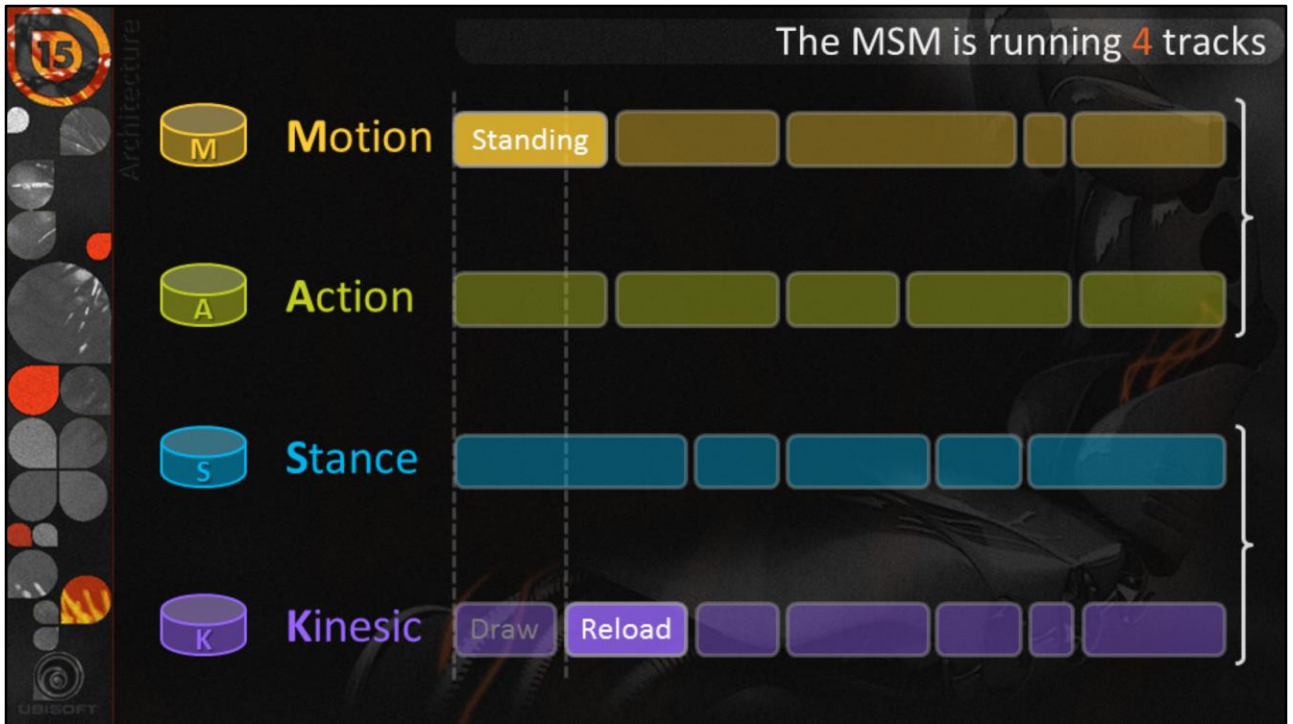
**Stance & Kinesic** are sharing another layer on top of the full body one within the animation engine

This structure is highly inspired on how humans actually think...moving efficiently and doing an action efficiently is not something that fits well simultaneously

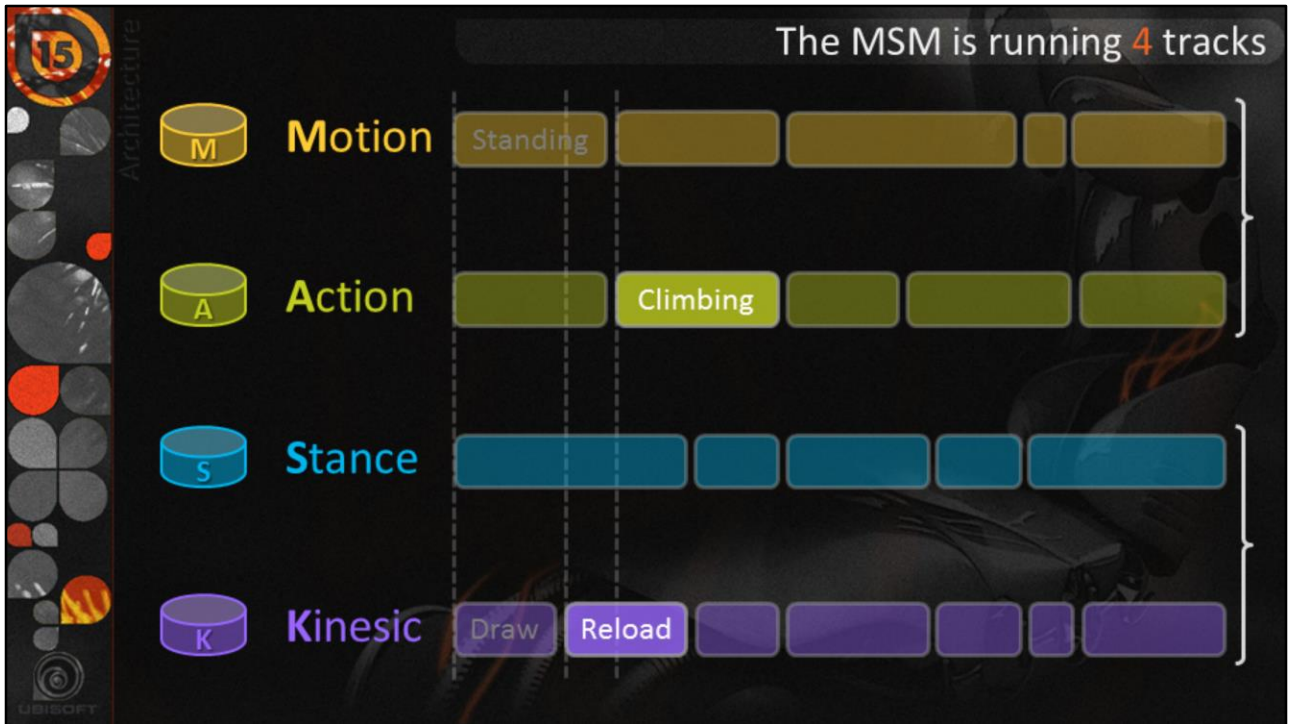


Let's have a look at a simple sequence example illustrating the inter track roles & services





Let's have a look at a simple sequence example illustrating the inter track roles & services



Let's have a look at a simple sequence example illustrating the inter track roles & services

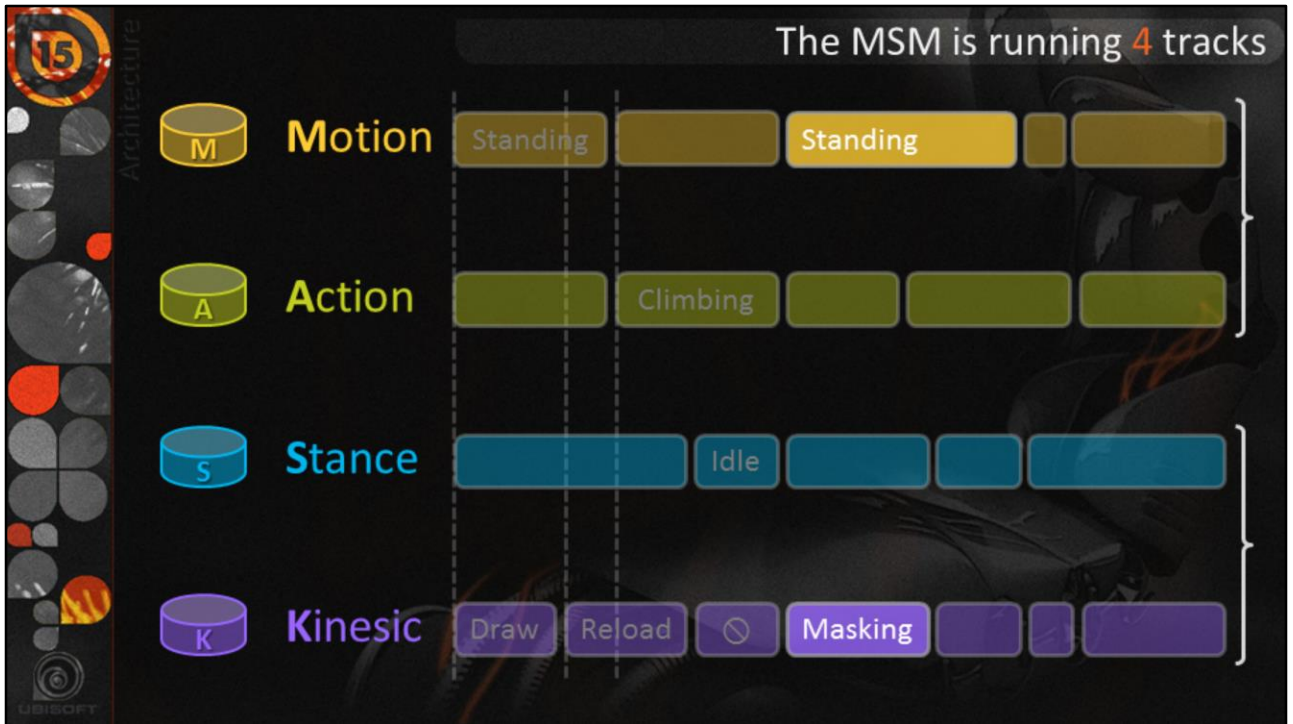




Let's have a look at a simple sequence example illustrating the inter track roles & services

Climbing decided at a certain point during it's climbing over animation to **disable anything related to Kinesic** since hands are being used.

This mechanism is a simple **bit mask** property that motions & actions can modify dynamically to allow or stop incoherent layered animations with them.



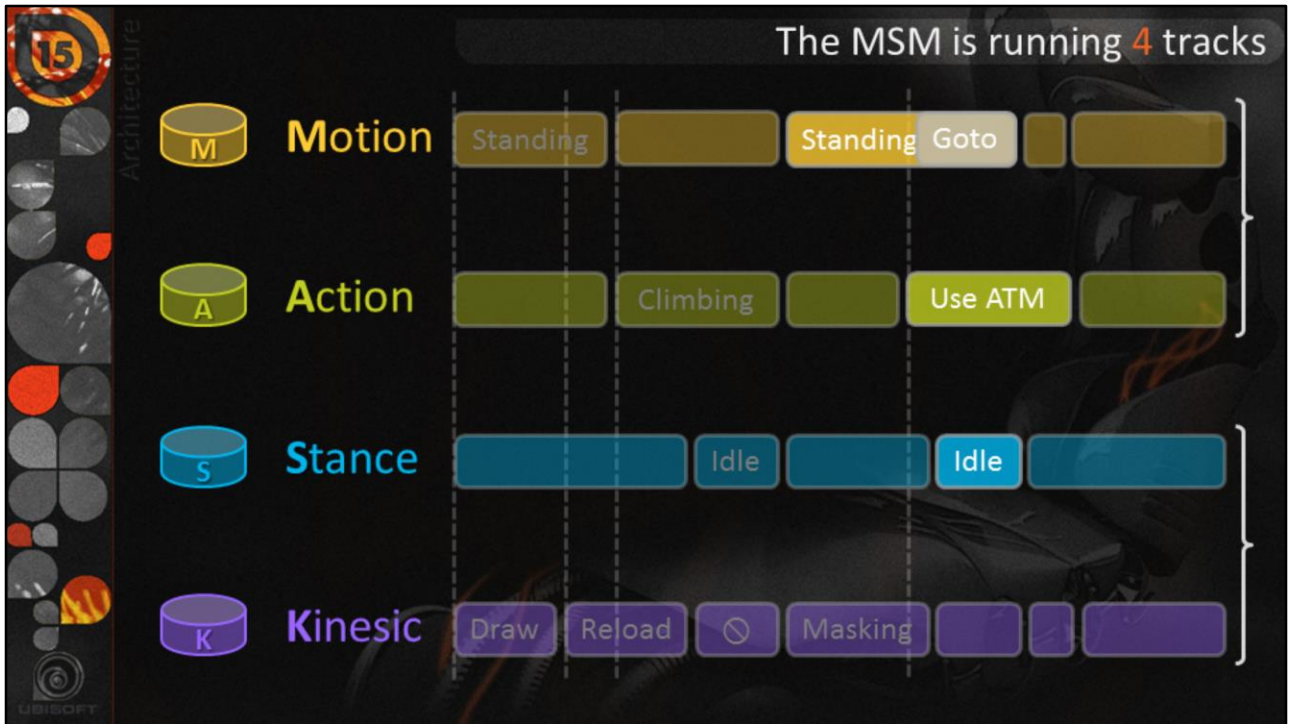
Let's have a look at a simple sequence example illustrating the inter track roles & services



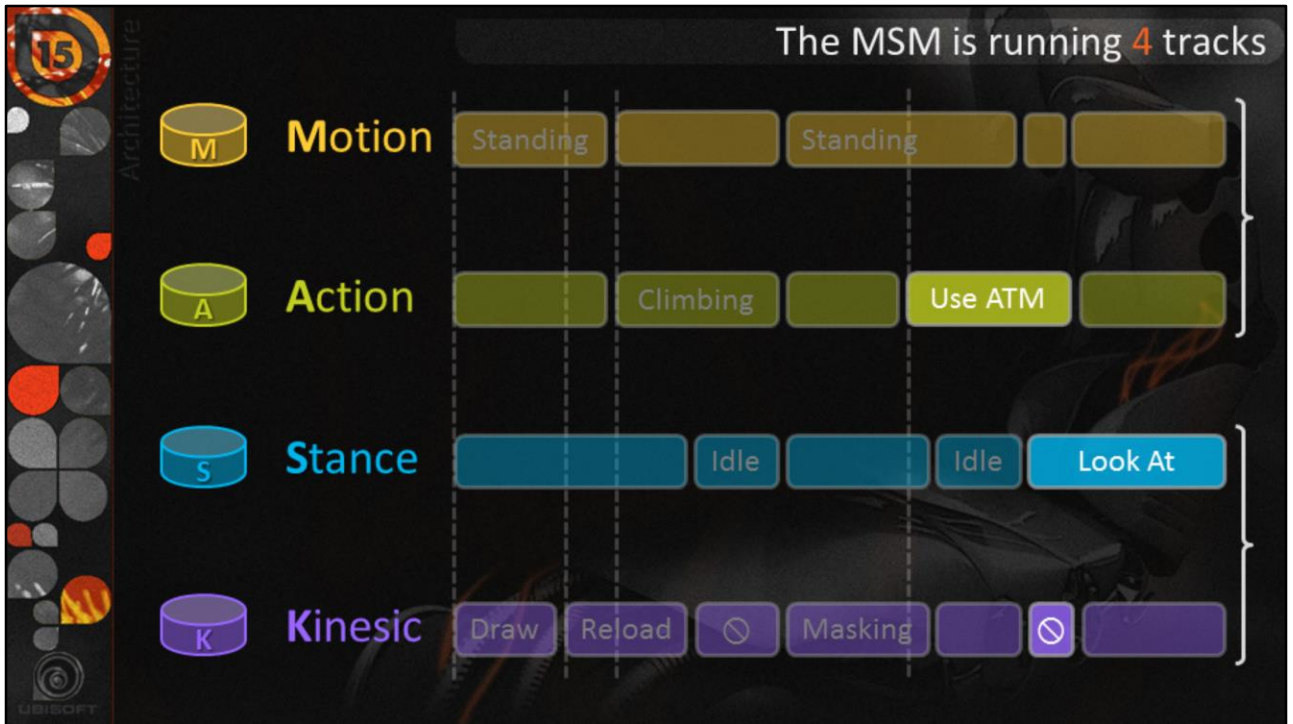
Let's have a look at a simple sequence example illustrating the inter track roles & services

ATM usage asked the Motion track to go to a specific location to start the interaction



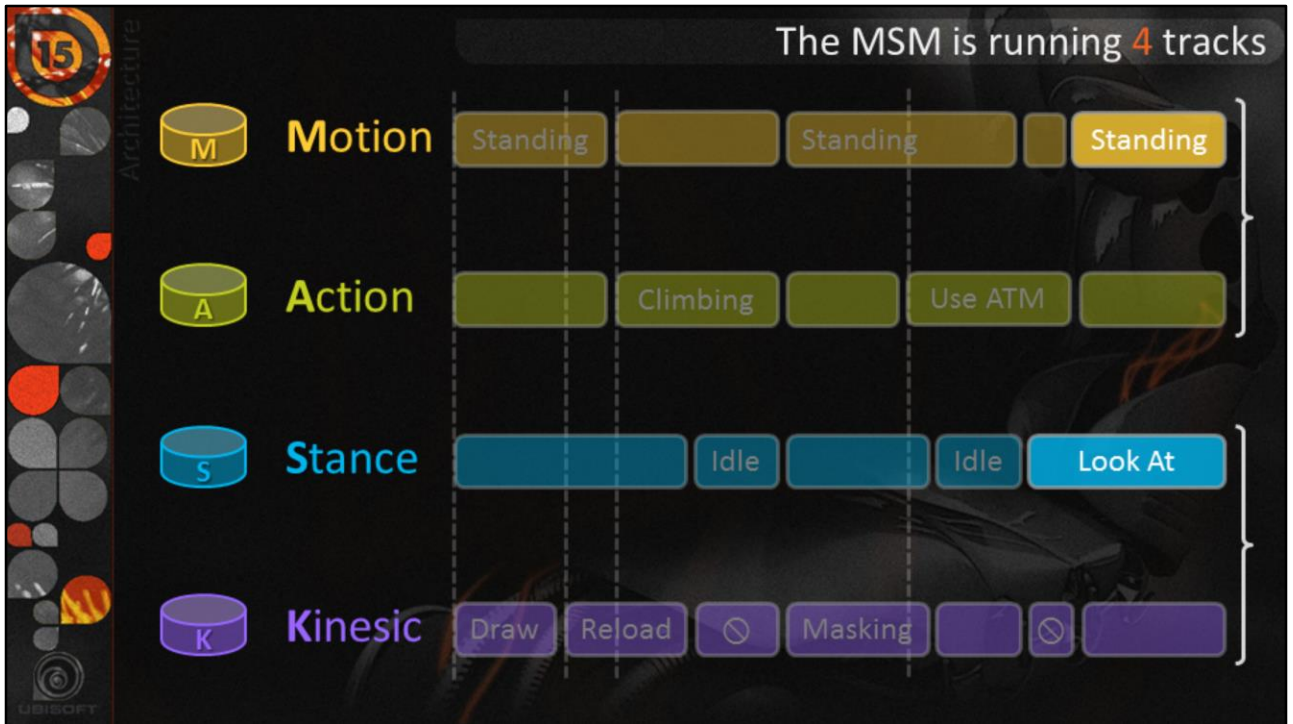


Let's have a look at a simple sequence example illustrating the inter track roles & services



Let's have a look at a simple sequence example illustrating the inter track roles & services

Again, **hands are be used** with the ATM so the bit mask removes any possible Kinesic states but at the same time, a **sexy lady** will pass by and trigger the look at state over the end of the ATM animation.



In between themselves, the top track of the pair can **overlay**, **pause** or **halt** the underlying one

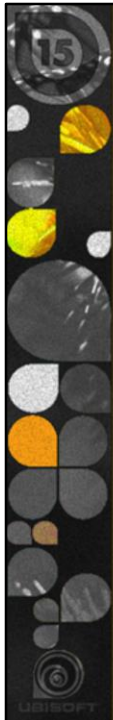




The rest of the talk will be split into **4 sections** showing some concrete examples of a couple of states of each of these tracks.

Let's start with the **Motion** track which is supporting the **usability** aspect of the **game feel**.

**Keywords** to keep in mind while building these states are **fluidity** and **precision**!



## Root motion approach for fluidity

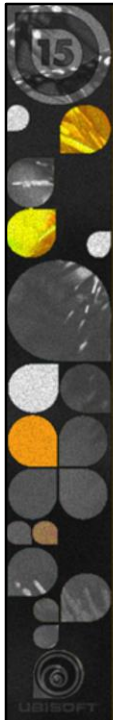
### Animation driving displacement

- mostly translations
- sometimes rotations



To enforce the maximum **fluidity** of movement possible we will use a **root motion** approach based on a heavy load of **motion capture** data.

Animations are going to drive the **collision displacement mostly in translation** because we will want to keep the **rotation for the code** side since the shape is pretty much cylindrical



## Root motion approach for fluidity

### Animation driving displacement

- mostly translations
- sometimes rotations

### Skeleton smoothed collision

- ~~cylinder shape~~ (no auto step)
- ~~capsule shape~~ (no high angle step)
- pencil shape



Speaking of the displacement collision, we tried various approaches like a :

- **Cylinder** shape, which was not good for **auto-stepping sidewalks**
- **Capsule** shape, which was not good for **steep steps** to climb
- Pencil like shape, which was good for most cases but **slightly glitchy sometimes with lateral noise on sidewalks** but it was compensated by putting a **spring damping** between the **root** of the animation skeleton and the **displacement** location

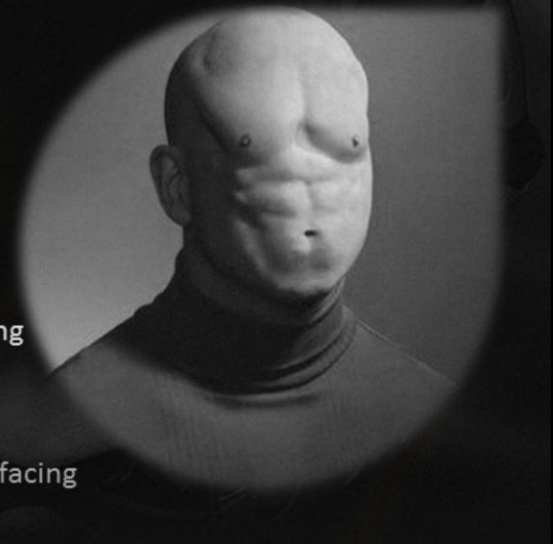


## Direction segmentation for fluidity

### Heading is the intention

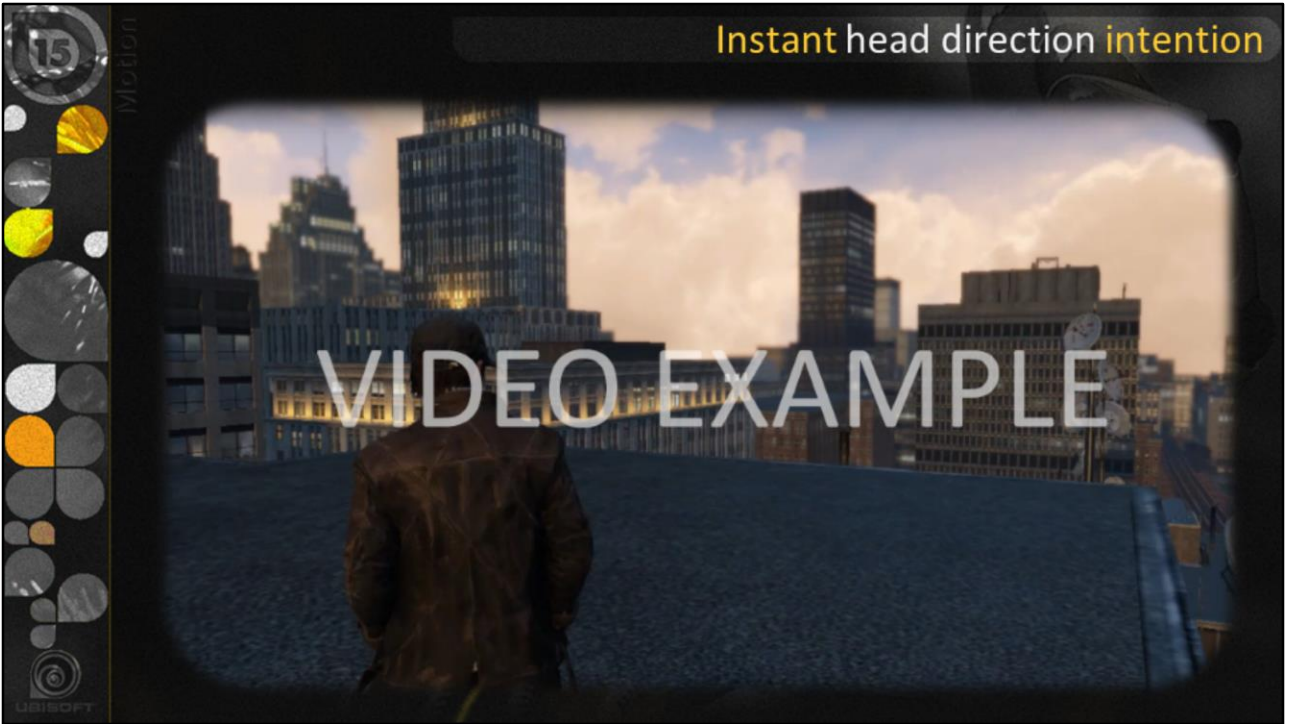
- expressed by head direction
- read from angular input
- high frequency filtering

### Facing is body lagging

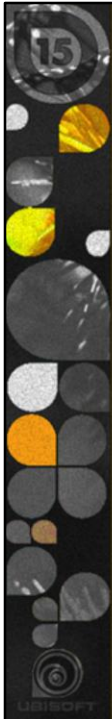


We said earlier that root motion rotations were handled by the code, mostly because we wanted to decouple them into a chain :

- **Heading** : **optical illusion** given by the **head** fast rotation towards the direct input of the joystick
- **Facing** : the rest of the **body lagging** behind with a sequential **chain of dampers**



Here you can see the **head leading** the movement being **almost 1 to 1 with the joystick** direction and then the **body will catch up** afterwards to keep everything fluid with some **natural inertia**



## Displacement **sliding** for responsive **precision**

### **Facing** is body lagging

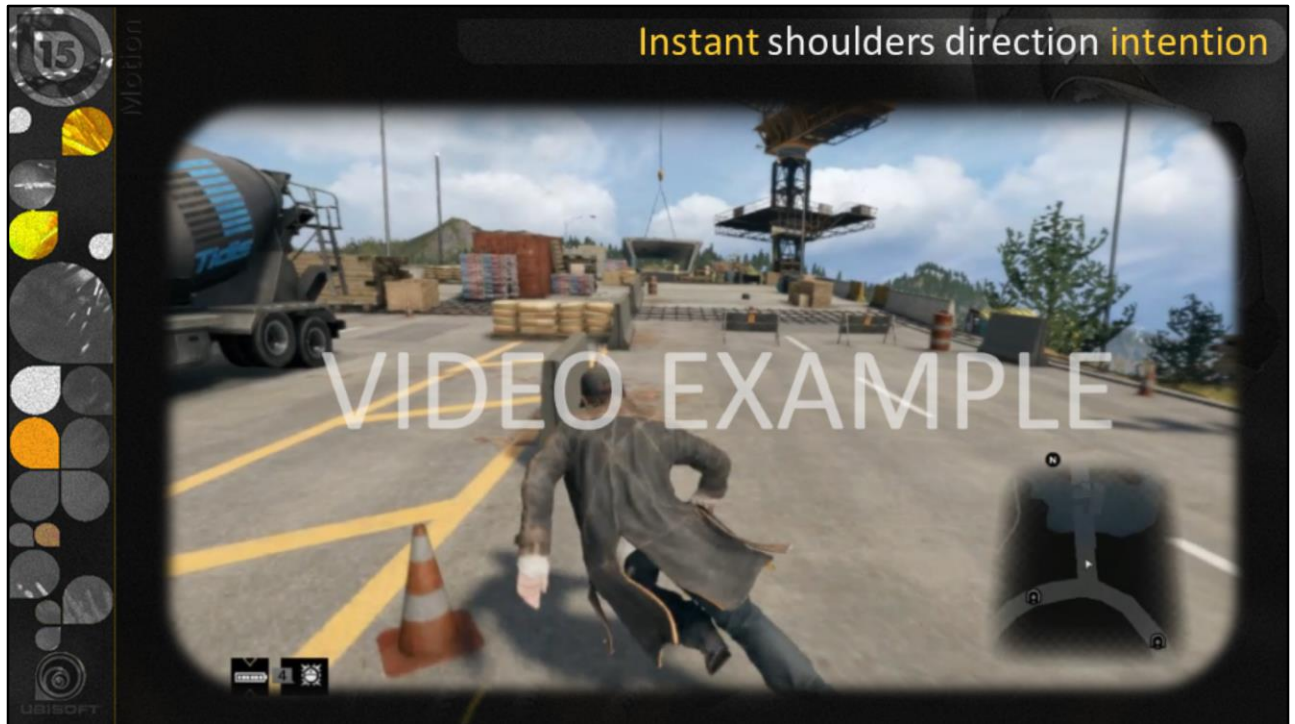
- expressed by shoulders direction
- read from lateral input
- high frequency filtering

### **Displacement** is sliding **input** heading

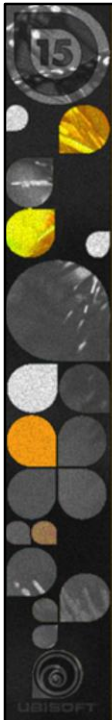


We said earlier that translations were going to be handled totally by animations but we lied...There a case here it is useful to make the displacement slide artificially :  
With facing lagging means we can read the **angular acceleration** of the joystick to **inject some lateral movement** to the displacement to dodge something and give the **optical illusion of the shoulders leading** the movement hiding completely the feet sliding on the ground





With **abrupt joystick inputs** given we can move the displacement towards the desired direction and on top of that we can use a **multi-dimensional parametric blend** to fit the movement properly



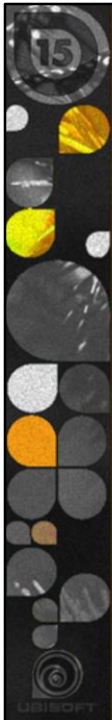
## Data markup for responsive precision

- Custom entry in animations
- Adaptive pose matching
- Latency reduction

*Movement :*

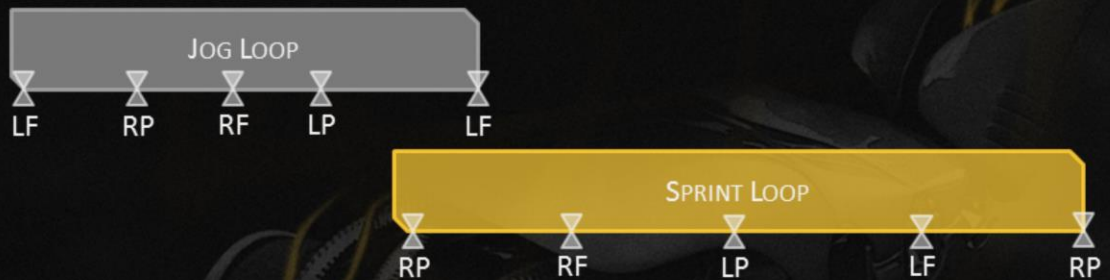


To continue to talk about precision, let's talk about animation mark up. Let's take **feet mark ups category** for example which are generated by an **automatic detection algorithm** for LF, RP, RF & LF. The key here is to **never wait for the proper mark up** to happen, you have to blend as soon as it is needed for the player.



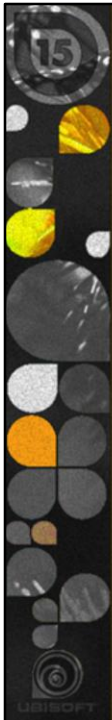
## Data markup for responsive precision

- Custom entry in animations
- Adaptive pose matching
- Latency reduction



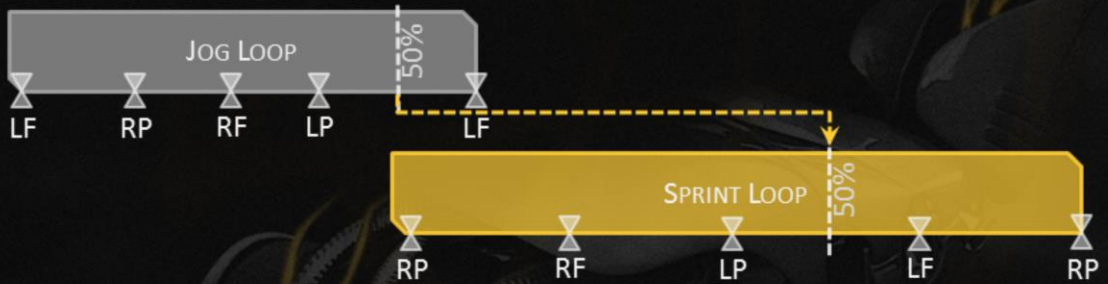
Let's say we are playing a **jog loop parametric blended animation** and suddenly the player wants to **sprint**.



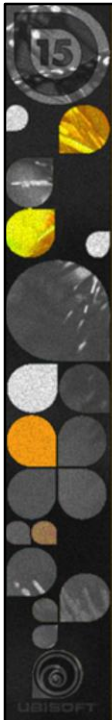


## Data markup for responsive precision

- Custom entry in animations
- Adaptive pose matching
- Latency reduction

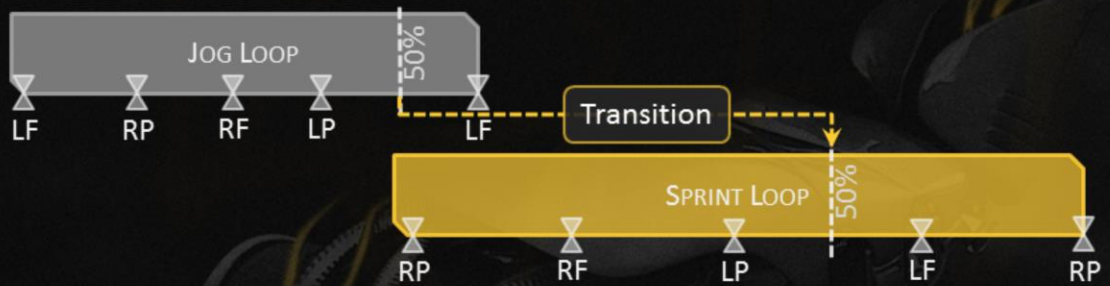


Let's use a **pose matching** to analyse where we are **between the main feet poses** to match the same pose pair in the following animation. That way we can **score every potential pairs** about their **proportion** and find a winner spot to blend in.

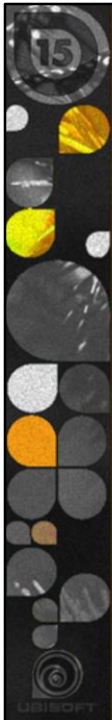


## Data markup for responsive precision

- Custom entry in animations
- Adaptive pose matching
- Latency reduction

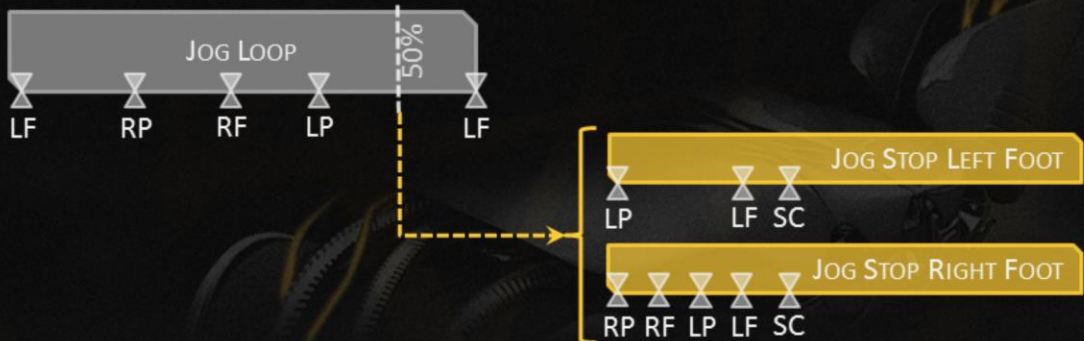


This can also be **applied to transitions** animators are going to plug in between specific loops to enhance the **feeling of inertia and speed build up**. We are going to pose match the transition first to then pose match the future animation



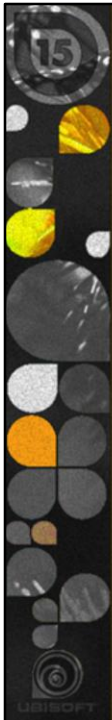
## Data markup for responsive precision

- Custom entry in animations
- Adaptive pose matching
- Latency reduction



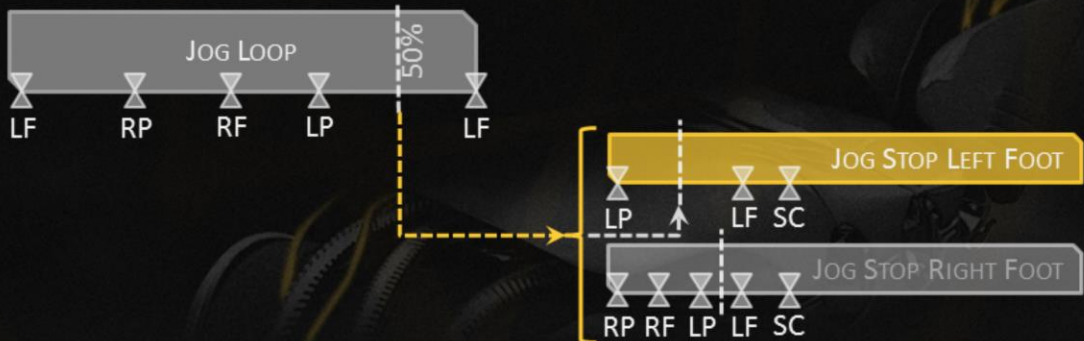
Pose matching is also used between loops and stop animations for example. In this case, we will have multiple stop animations to choose from. One starting with the left foot and the other starting with the right foot. The **scoring function** will consider the **good pair**, the **proportions** of this pair and the **frame location** of this pair also to choose a winner.





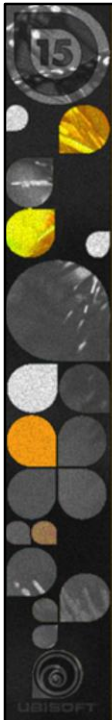
## Data markup for responsive precision

- Custom entry in animations
- Adaptive pose matching
- Latency reduction



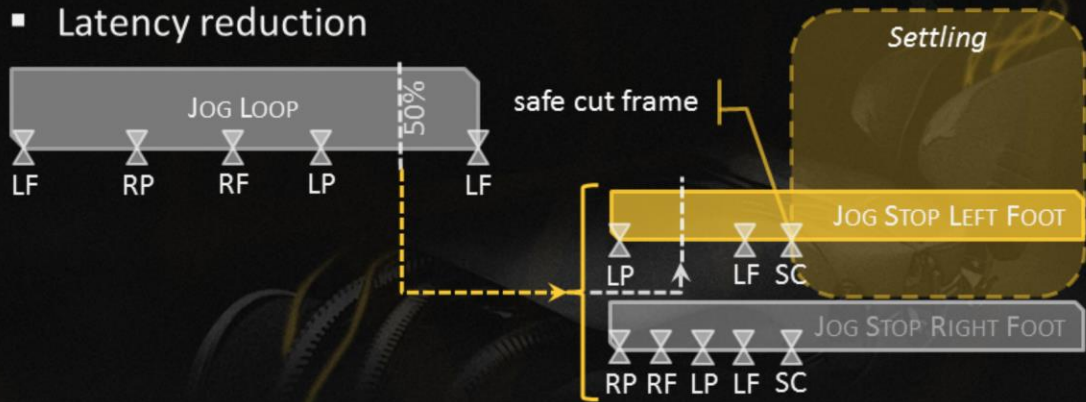
Pose matching is also used between loops and stop animations for example. In this case, we will have multiple stop animations to choose from. One starting with the left foot and the other starting with the right foot. The **scoring function** will consider the **good pair**, the **proportions** of this pair and the **frame location** of this pair also to choose a winner.

On top of that, you can **extend the pose matching to multiple pose categories** like **arms mark up** or other special mark up you want. In the end, you will have to **tweak** the accumulation of these categories' **scores weight** to select the proper animation to play.

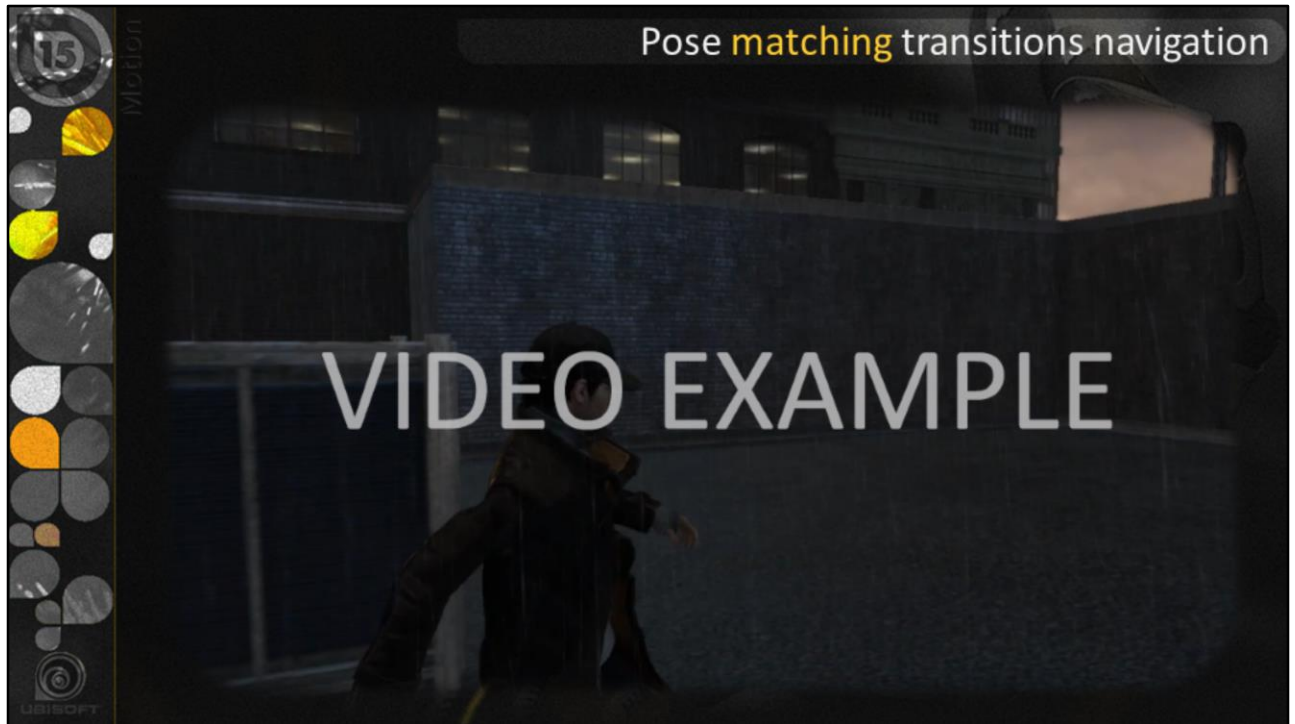


## Data markup for responsive precision

- Custom entry in animations
- Adaptive pose matching
- Latency reduction

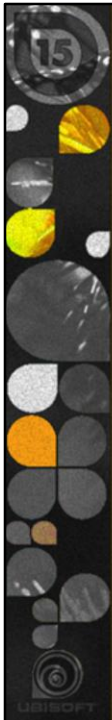


We are often breaking animations to play another but there are some sweet spots to add some acting on the character sometimes. That's why most non-looping animations will have a **settling portion after the crucial portion** and this separation will be defined by the **safe cut mark up**. That way, we can give some contextualism to the character fitting with what previously happened and also say to the code that from now on, this animation is 100% interruptible by anything.



Let's have a look at various **speed changes** using **pose matching** through **data driven transitions**



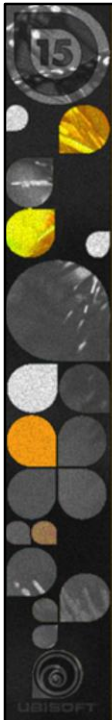


## Data segmentation for start precision

- Optical illusion : Head motion reactivity
- Read mind for direction & speed : Fixed delay
- Anticipation of ~6@8 frames (*playtest*)

START

Finally, to boost the precision and the responsiveness of the start animations we will use the trick of the head motion reactivity because we don't know when the player starts to push the joystick **at which force and in which direction it will end**. We can't read the mind of the player...so playtests showed we needed between **6 to 8 frames** at 30 fps to know if the start animation will finish to **idle for a turn on spot, to walk, to jog or to sprint**.



## Data segmentation for start precision

- Optical illusion : Head motion reactivity
- Read mind for direction & speed : Fixed delay
- Anticipation of ~6@8 frames (*playtest*)
- Same philosophy applied to plants & turns



So we separated the start into 2 sub states :

- One for the **common anticipation** of all the endings possible : **moving mostly the head**
- One for the **desired ending** at the **correct speed** and the **correct angle**

The same concept has been applied to **plant & turns** to **be able to anticipate** and to **be reactive**

The **same approach** was used when we built the **cover system**



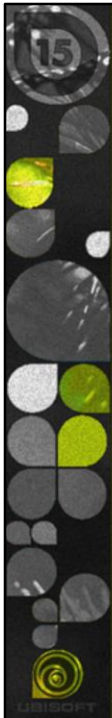
Here we can see various examples of starts & plants **triggered and interrupted** with **various endings**





Let's follow with the **Action** track which is supporting the **grounding** aspect of the **game feel**.

**Keywords** to keep in mind while building these states are **seamless** and **contextual**!



## Dedicated sensors for contextual climbing

### Physical **shape/ray** sensor

- Real-time & Functionnal, but...
- Expensive on many characters

### Edge **annotation** sensor

- Offline generated & "Precise"
- Cheap with some filtering

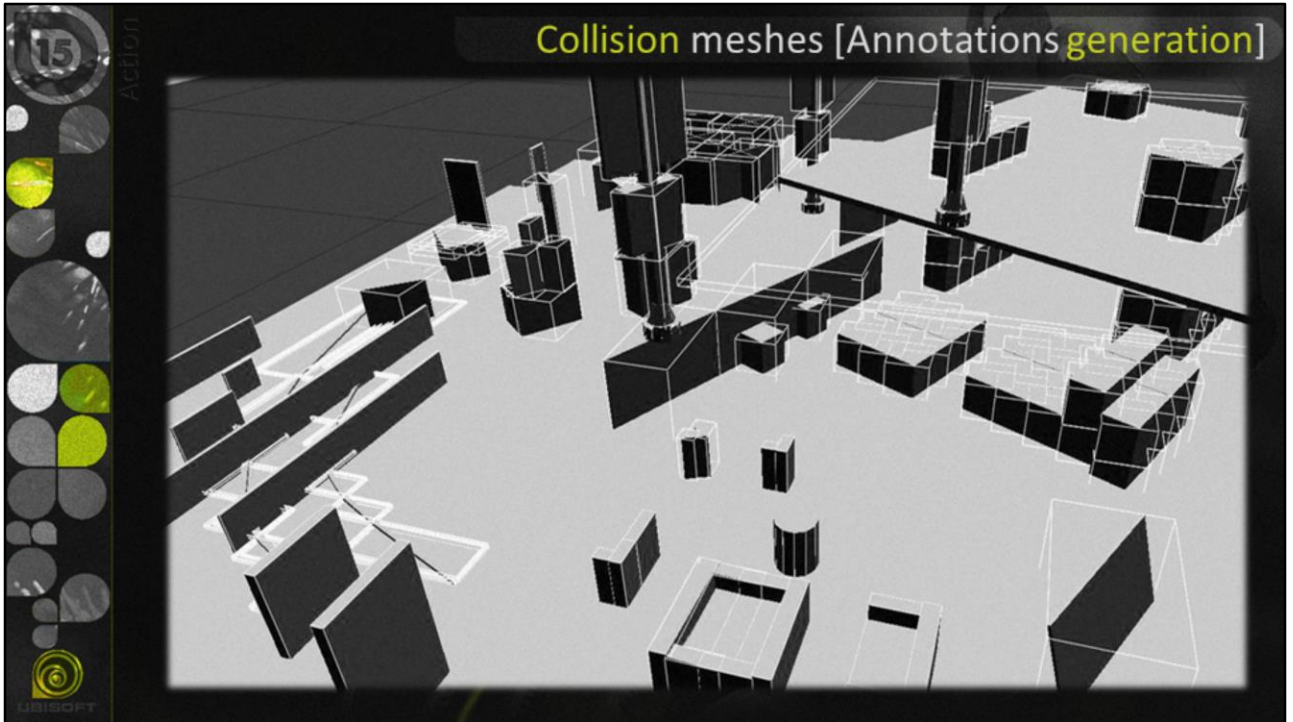


The **first sensors** developped were **shape and ray casts**

- Functional, but can be hard to entirety obstacles properly
- Expensive if used by many characters

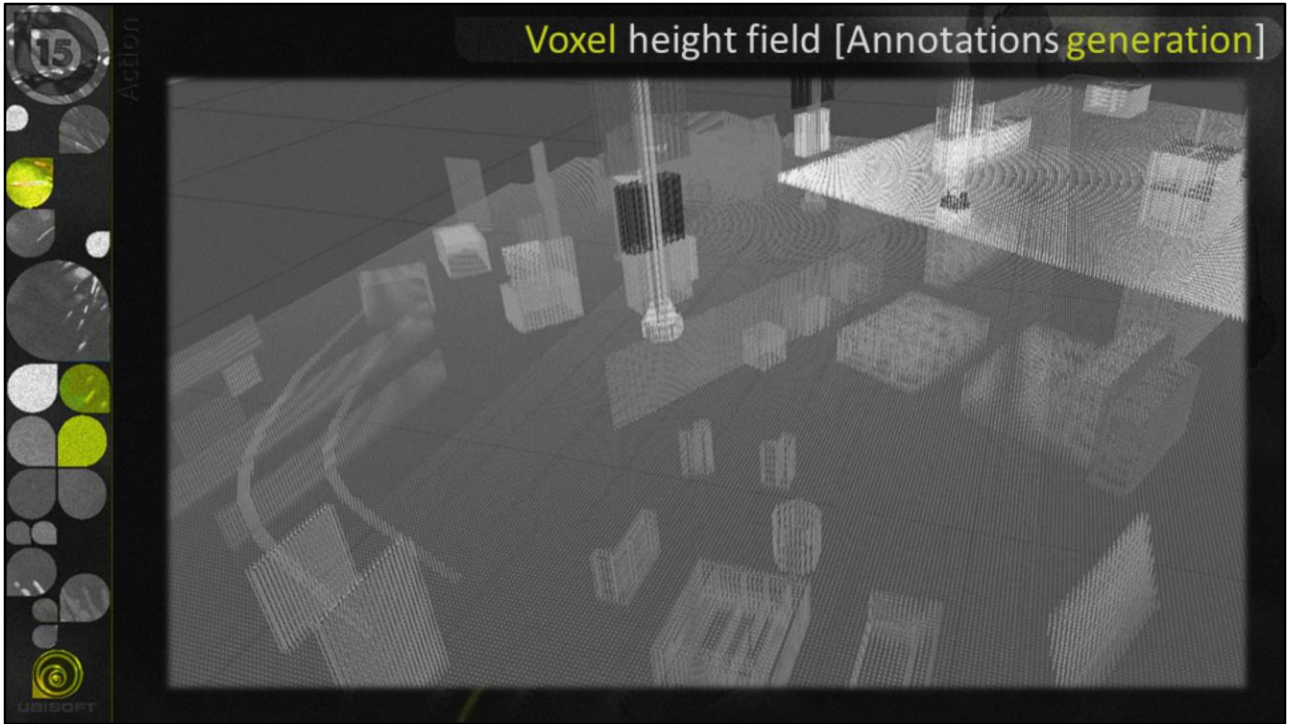
The **second sensors** developped were **offline data generated** ones

- Scan game world to detect ledges of every static object
- Same information used on static and dynamic objects

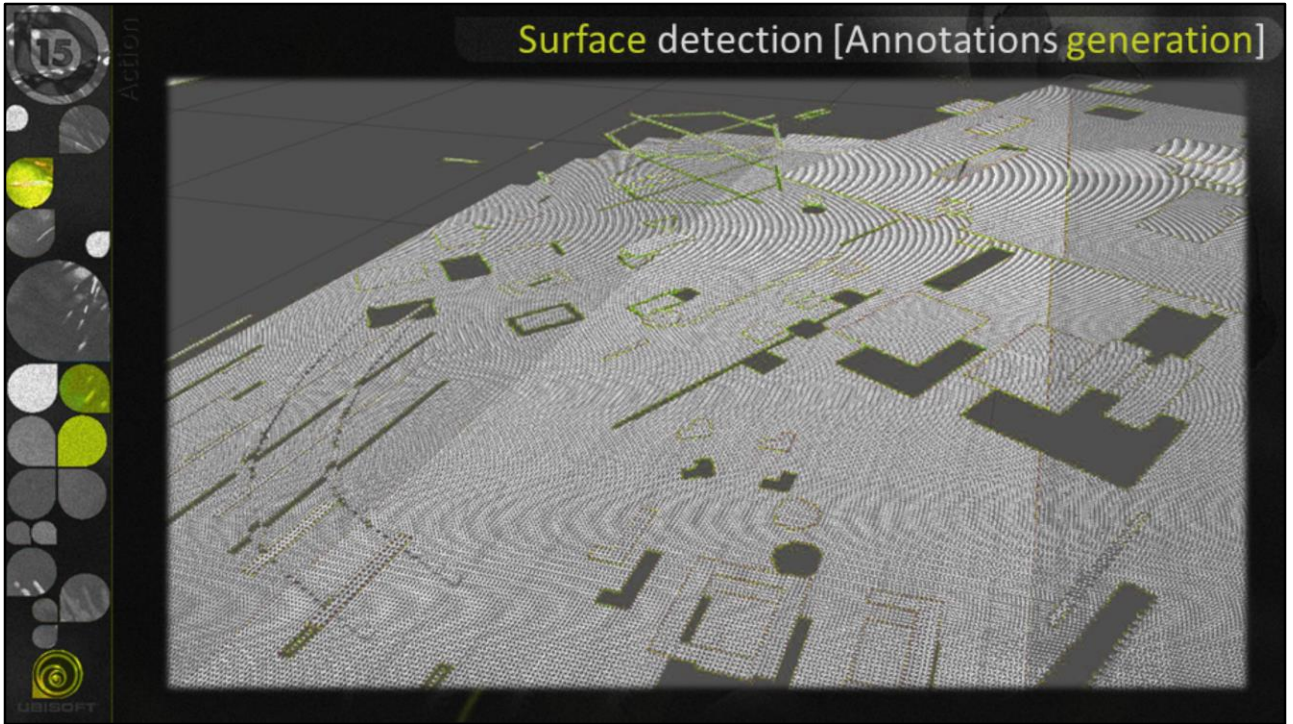


These offline generated annotations are built from the **collision meshes in the static world**

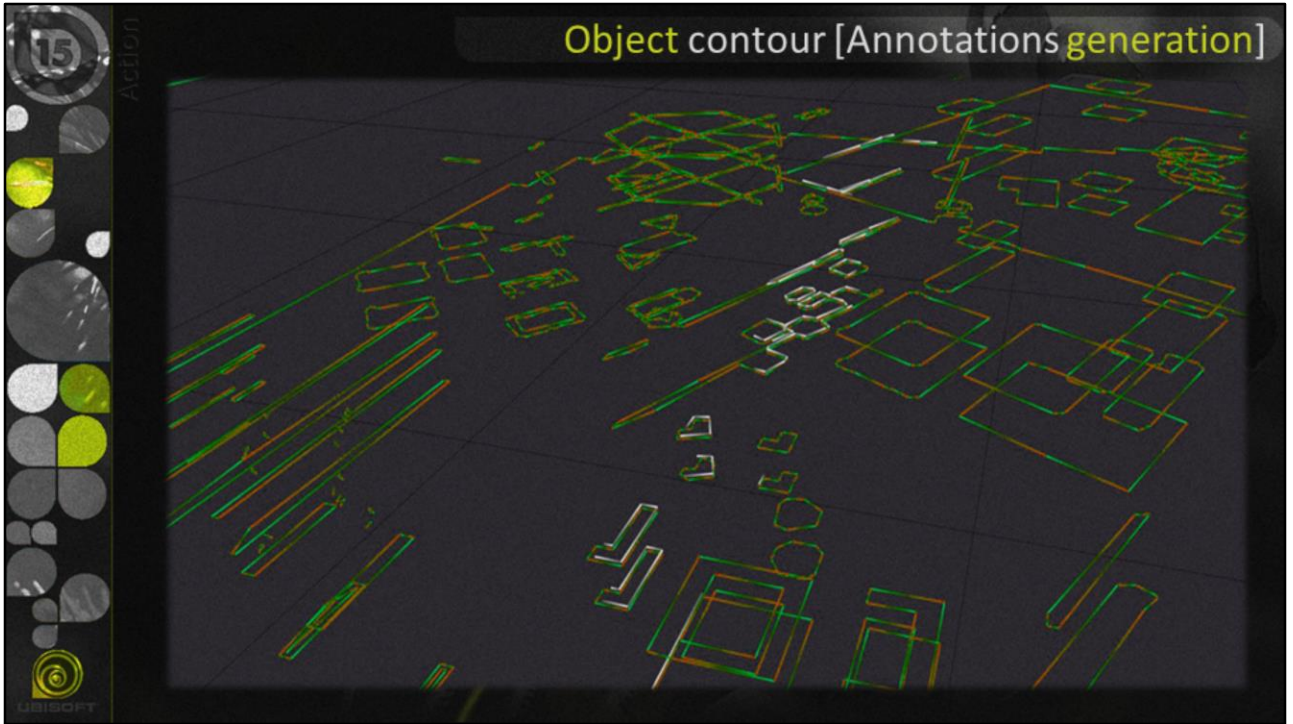




To afterwards do the voxelization of the world to build a **voxel height field**

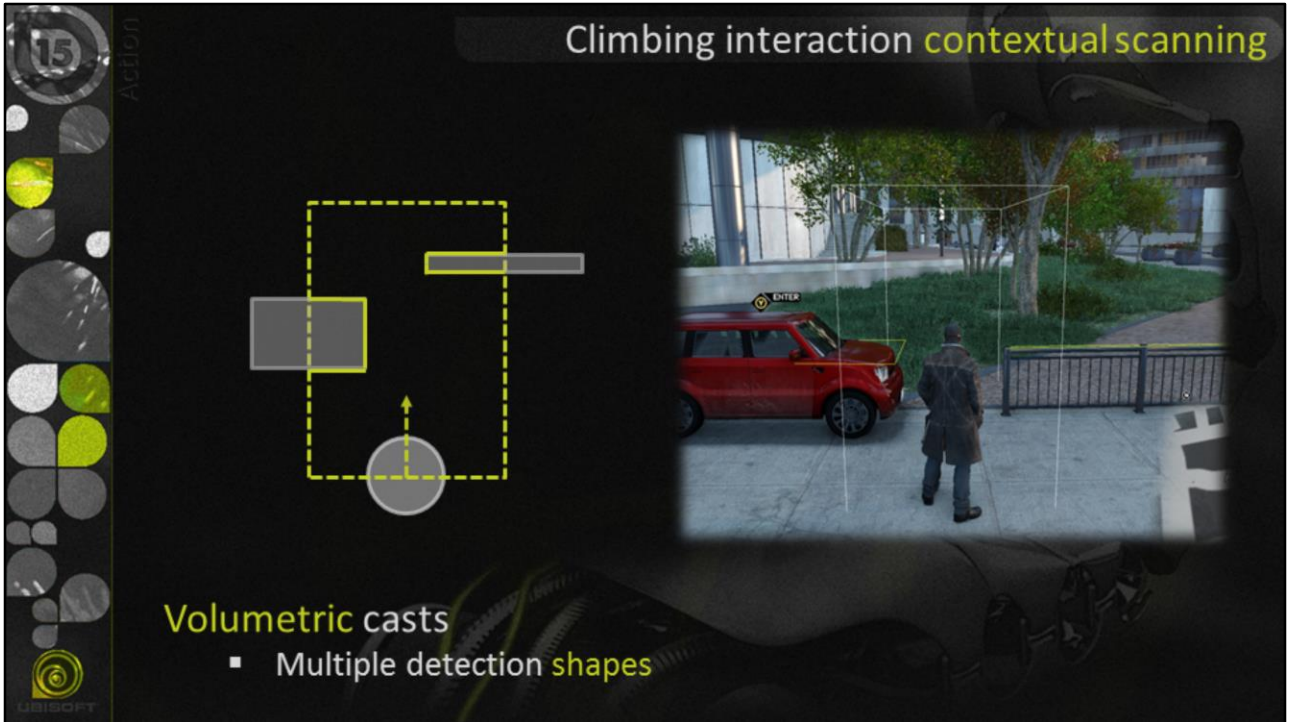


From which we will extract surfaces with proper **filters** and proper **thresholds**



To obtain in the end all useful **object contours** from which we will create the annotations

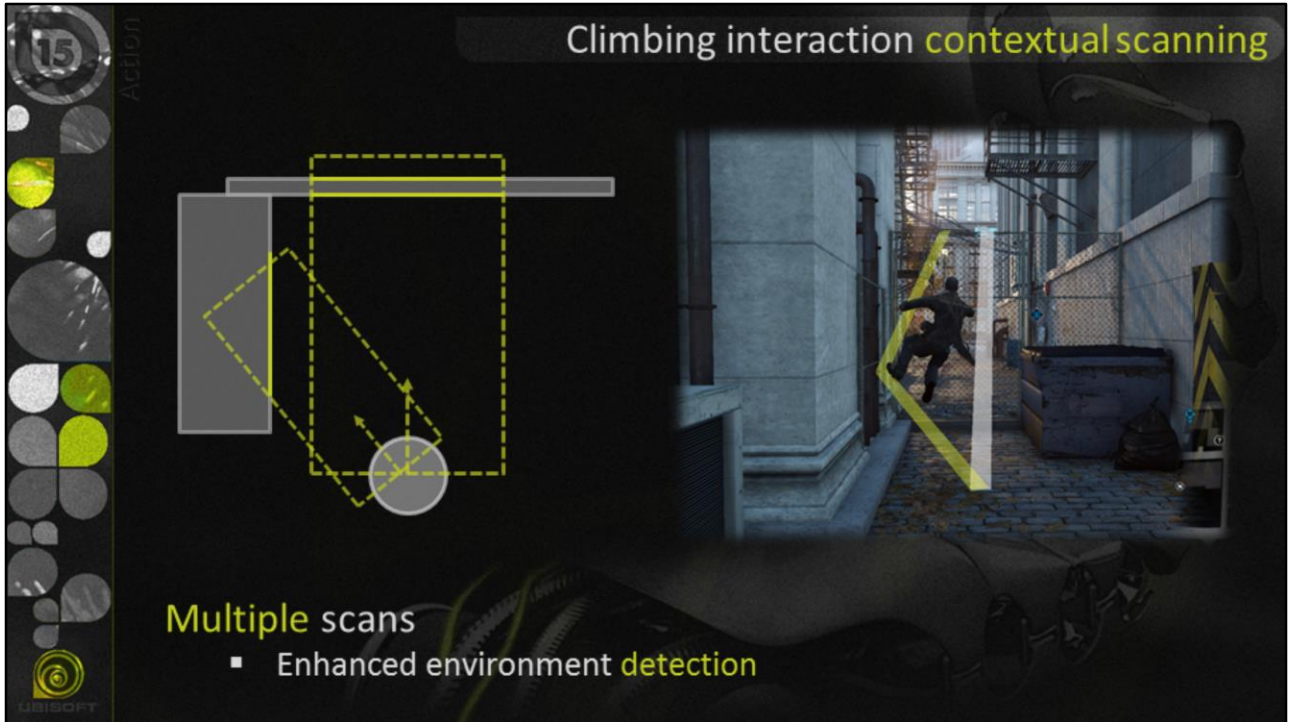




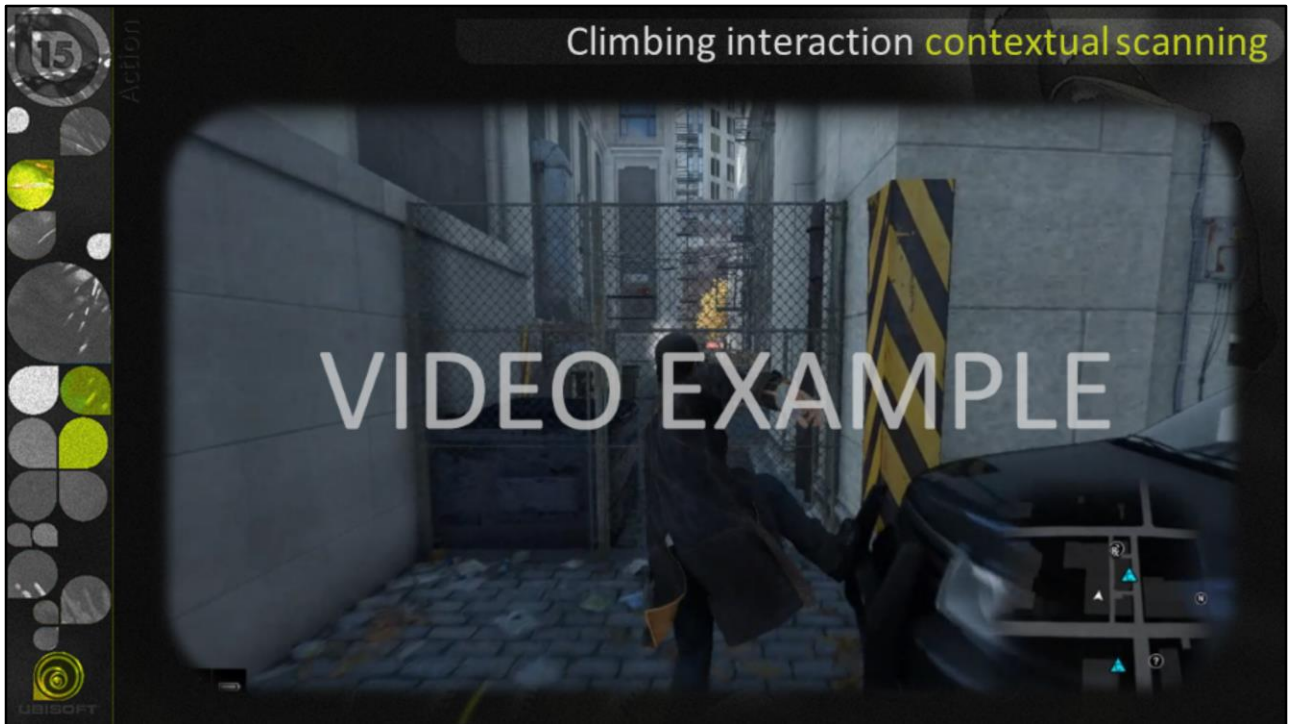
All these annotations are going to be **stored into a spatial hash map** for optimal search time since we have **thousands** of annotations per world sector. On top of that, we will have **dynamic objects** adding their own annotations in local space to the world

The **first advantage == Performance**

to avoid having too much data overlapping and check clearance for the character

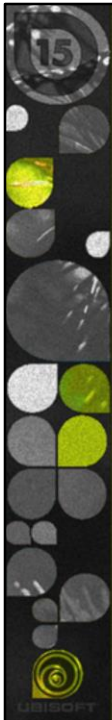


In the end the scanning was **so cheap** that added **lateral scanning to boost the contextualization** of jumping animations. That way we were able to **find walls or objects** on the side the support the climb over animation in that case.

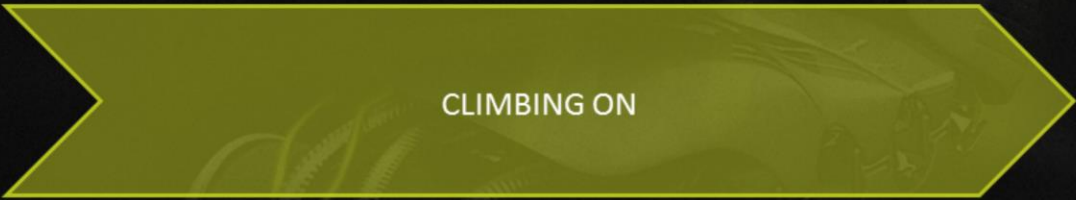


Here is a quick look at how it behaved in-game for a fence **surrounded** by **static walls** or **dynamic objects**




Data **segmentation** for **seamless** sequence

- One single mocap clip : Seamless action
- One single animation scene : Easier editing
- Multiple sub-states logic for scalability



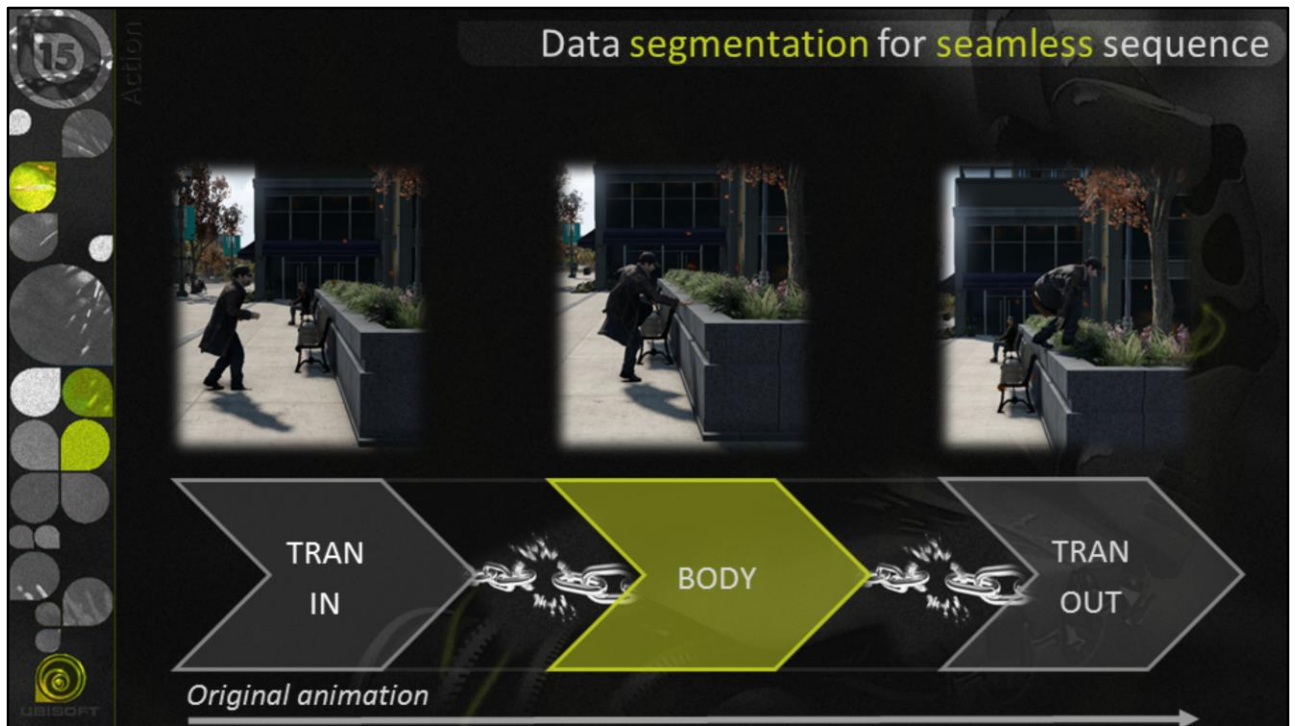
CLIMBING ON

*Original animation*



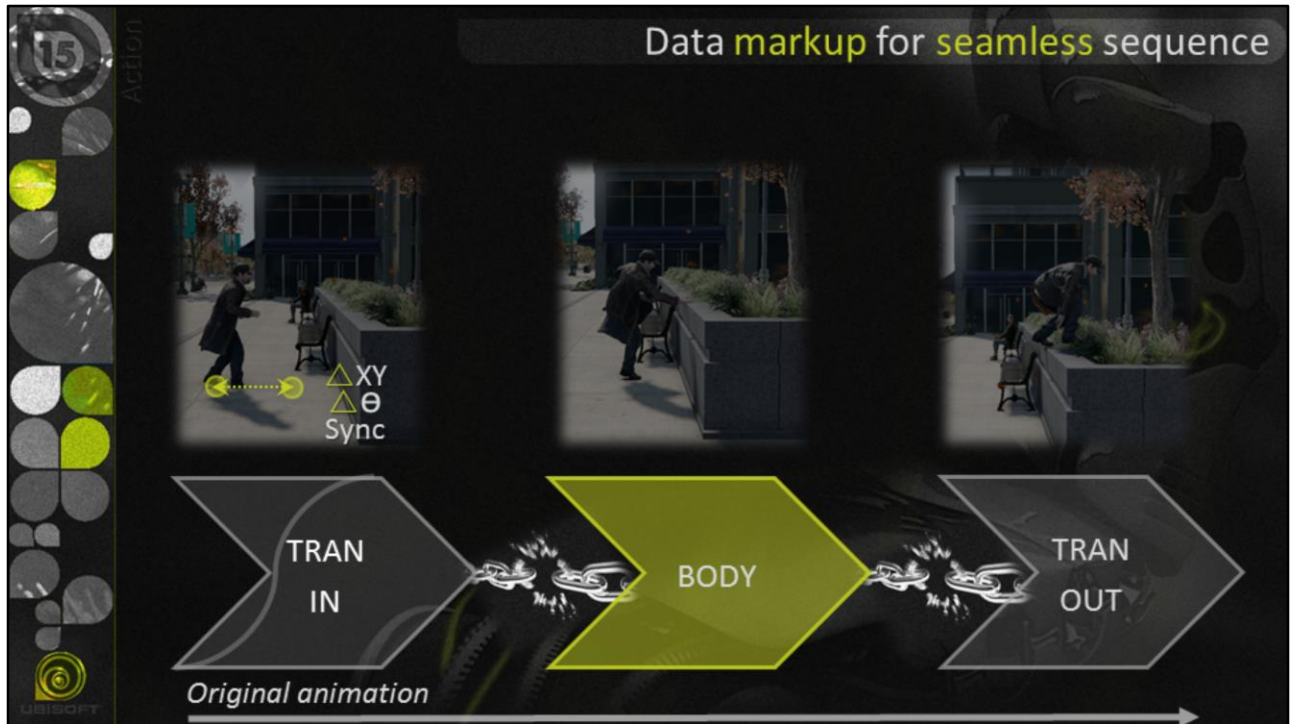
We previously talked about **data segmentation** in the motion track section and here again we will try to split the data to fit our needs in terms of **seamless playback** into the world.

For the climbing on animation for example, we will use **one raw motion captured animation** to ease the editing for the animator but we will split it into **3** different sub states for **logic purposes**.



For that example, it's split into 3 parts, the **transition in**, the **body** and the **transition out**

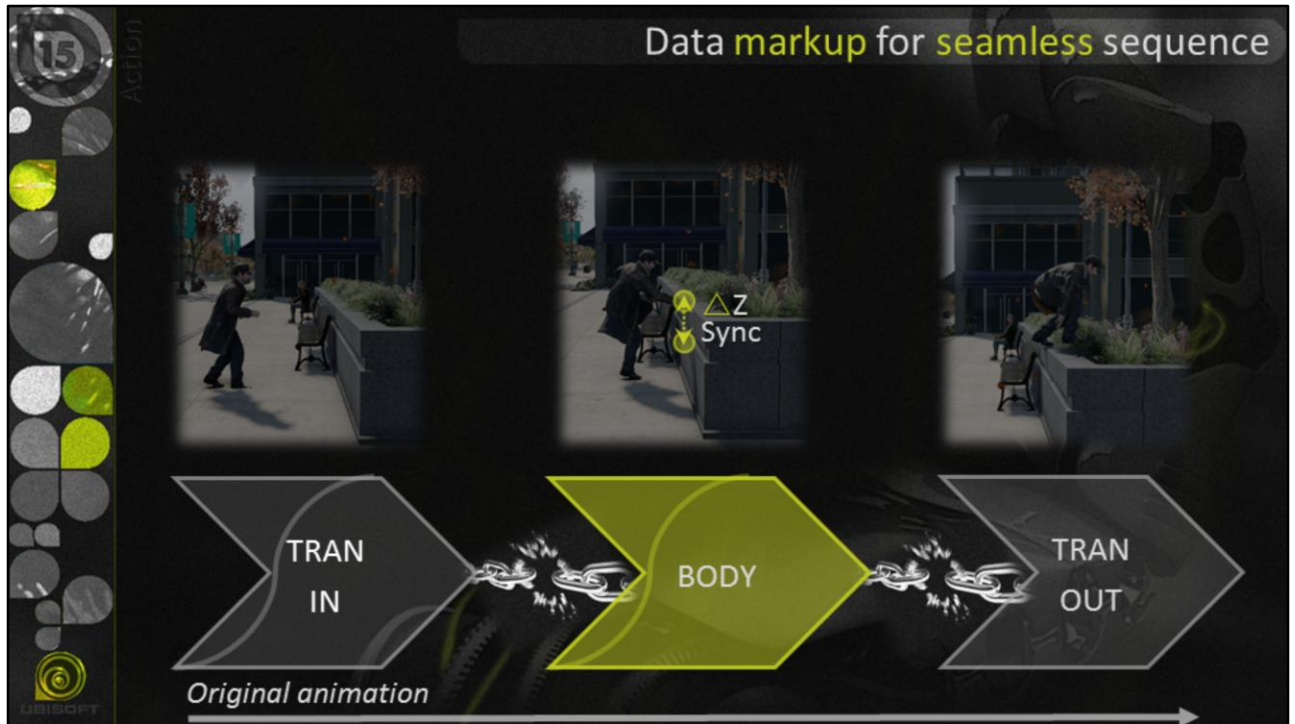
Some other jumping actions might add an **approach sub state** before the transition in or a **landing sub state** before the transition out



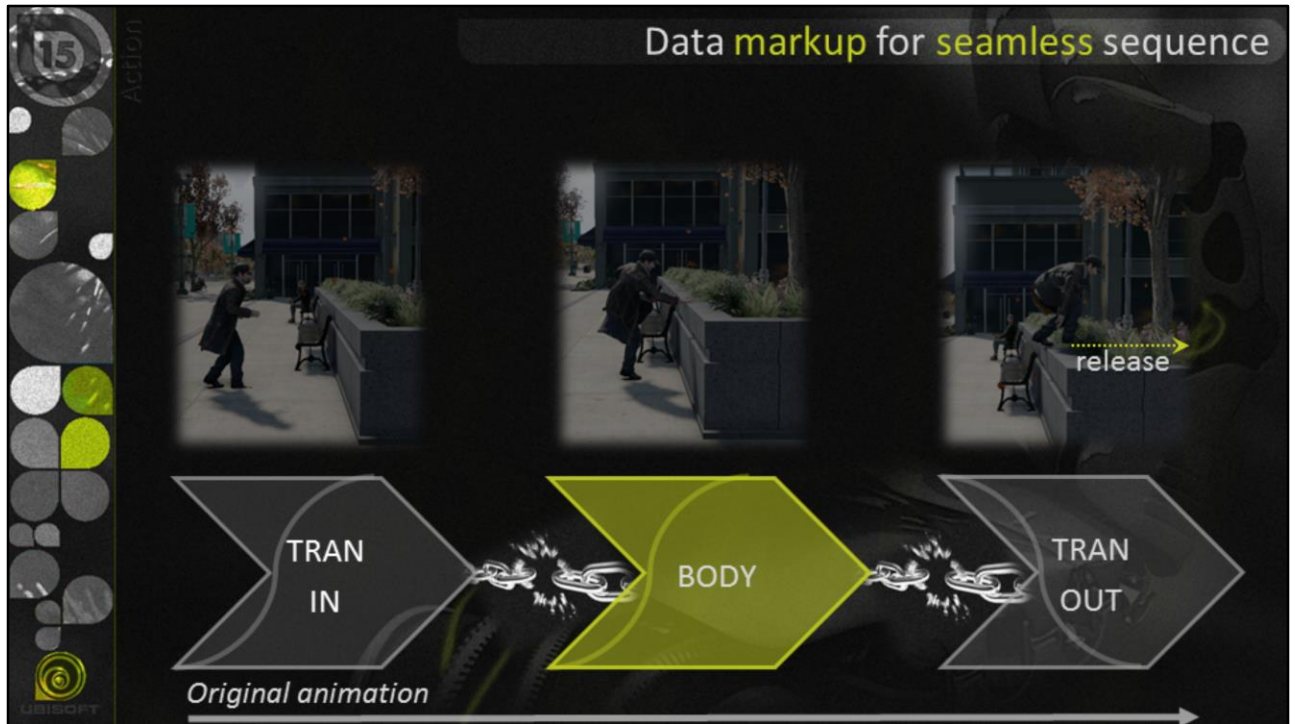
Each of these specific sub state will have a responsibility to make this single animation fit the imperfections of the world.

**TranIn** for example compensate the fact that the original animation was meant to start 1 meter away from the wall on the **XY plane** and it was also supposed to start totally forward without any angles. But the **mark up** in this animation will allow us to blend to this **synchronized anchor** which will compute the **delta** between the reality and the original location. This error, in translation and in rotation will be **distributed on all the frames** of the TranIn portion. But not in a linear way, it will inject this error compensation on each frame based on its relative **proportion** to the total displacement done during this segment. That way the animation will be fully sync to start the body segment.

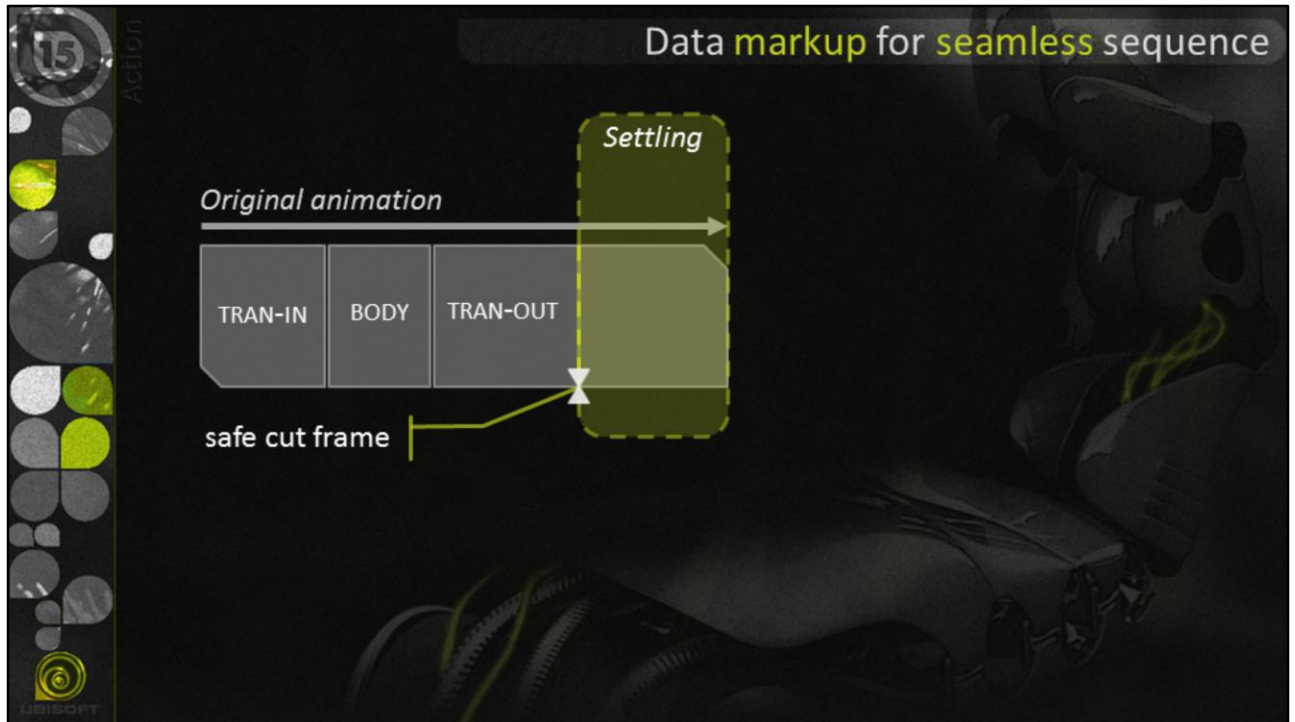




**Body** will then do the same but with the Z axis since the animation was meant for a **1m** high wall instead of a **1.3m**. During the whole segment, the error will be distributed to arrive flush at the top with the proper hand placement.



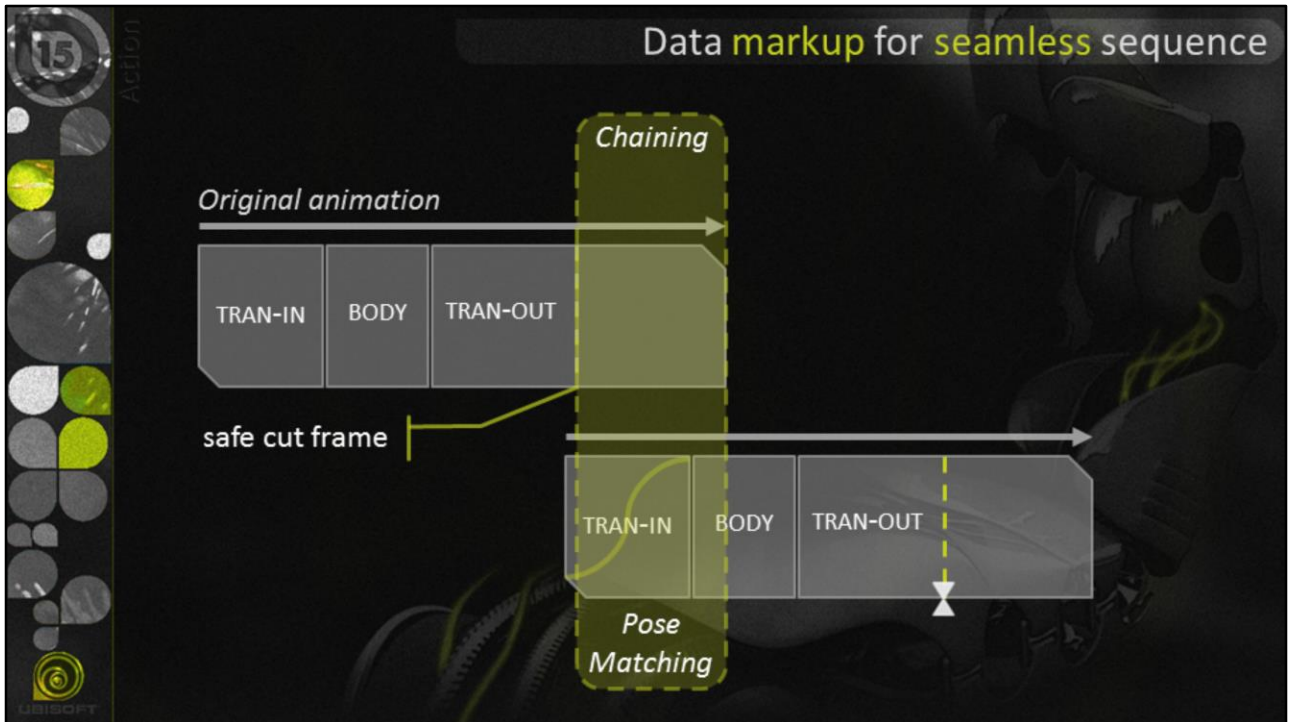
**TranOut fades out** all these synchronized anchors pretty rapidly to **give back progressively** as soon as possible the control to the player for its **direction** and **speed**



We briefly talked about the **settling portion** of the animations during the motion track part and here again, the transition out segments will be a good place to play some **nice acting** if the player is not touching the joystick.

But this **safe cut mark up** is also going to be useful to tell the code that it can be interrupted to **trigger another jumping action** since annotations could be pretty close to each other. So instead of settling, we will interrupt and trigger back to back the next one. We will reuse the **pose matching algorithm** to enter at the proper frame to be as seamless as possible.



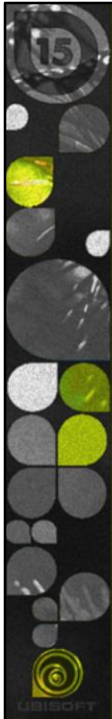


We briefly talked about the **settling portion** of the animations during the motion track part and here again, the transition out segments will be a good place to play some **nice acting** if the player is not touching the joystick.

But this **safe cut mark up** is also going to be useful to tell the code that it can be interrupted to **trigger another jumping action** since annotations could be pretty close to each other. So instead of settling, we will interrupt and trigger back to back the next one. We will reuse the **pose matching algorithm** to enter at the proper frame to be as seamless as possible.



Here is an example of how it looked in game when **chaining multiple jumping actions back to back...**



## Dedicated sensors for contextual reactions

### Stimulus sensor

- Catching impacts, explosions, etc...
- Managing reactions & impulses

### Body sensor

- Acquiring bone poses
- Computing body directions

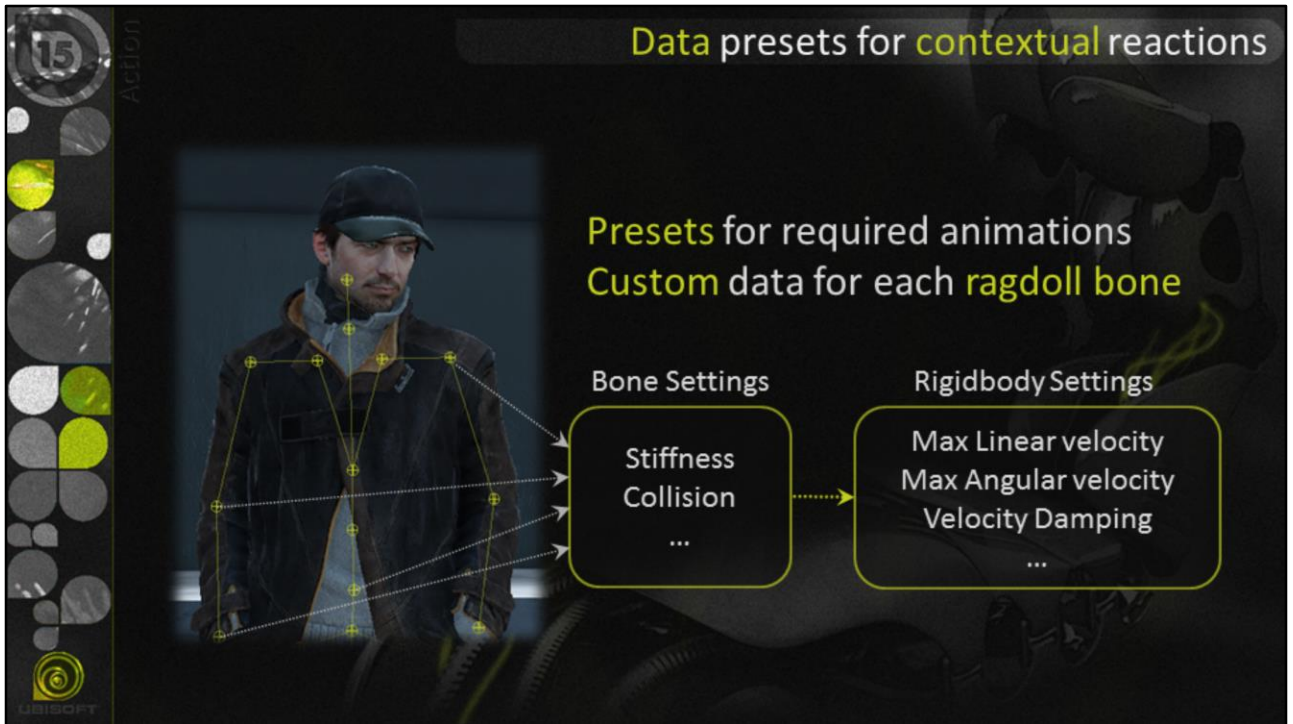


A second example for the action track will be the ragdoll reaction state

Which is using some **impacts sensors** to listen to **bullet hits, impacts, explosions** and stuff like that.

And with this, we are going to have some **body sensors** to know **how the body is placed, oriented and moving**.





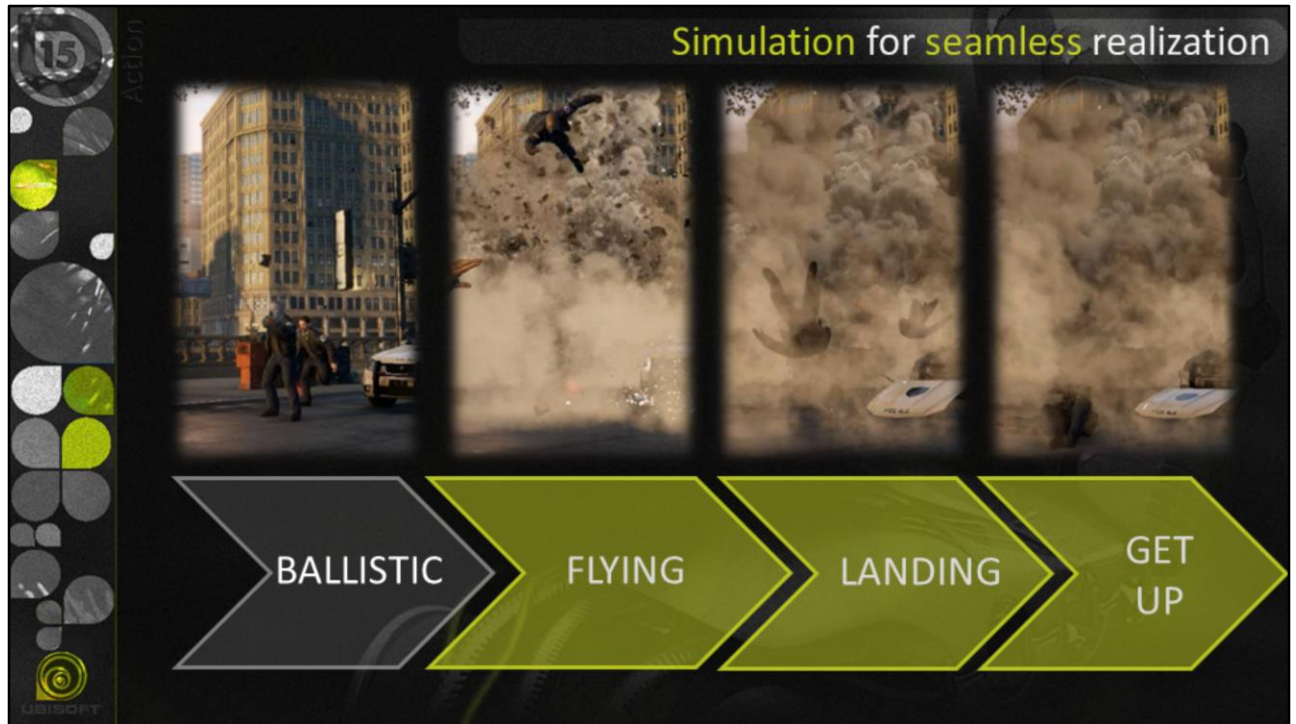
Using the **Havok** ragdoll, we built a **preset system** for **animators** to let them **tweak** properly the **ragdoll** for any of their animations with custom configurations :

- **Ragdoll type** for rigid body controller or powered constraint for example
- **Stiffness or collision** enabled of the bones
- **Velocities or damping** for the bodies

That way they **control better** what will be triggered when blending **from animation to physics**

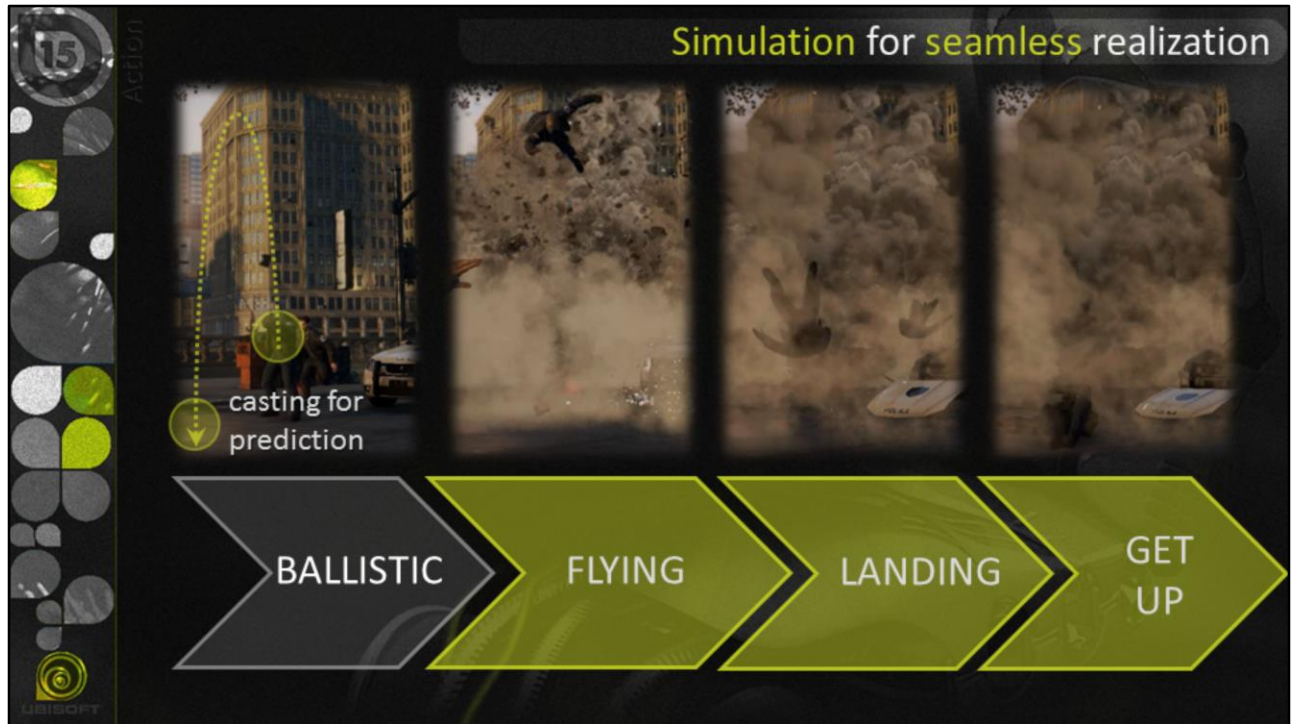


Ragdolls are mostly used in the industry for **projection in the air** and it can lead to **special organic** situations sometimes that are **difficult to tweak** to get an appealing result

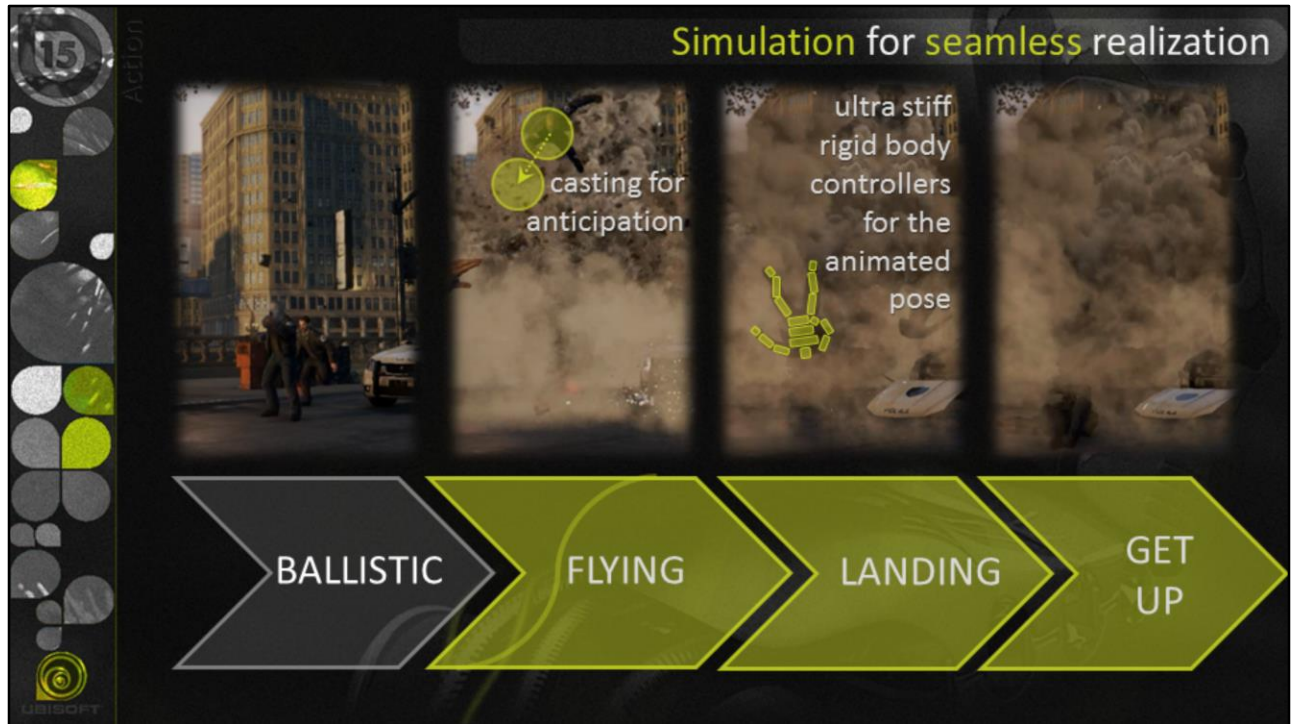


Instead of going **fully procedural**, we decided to go **heavily data driven** with a **trajectory system** instead



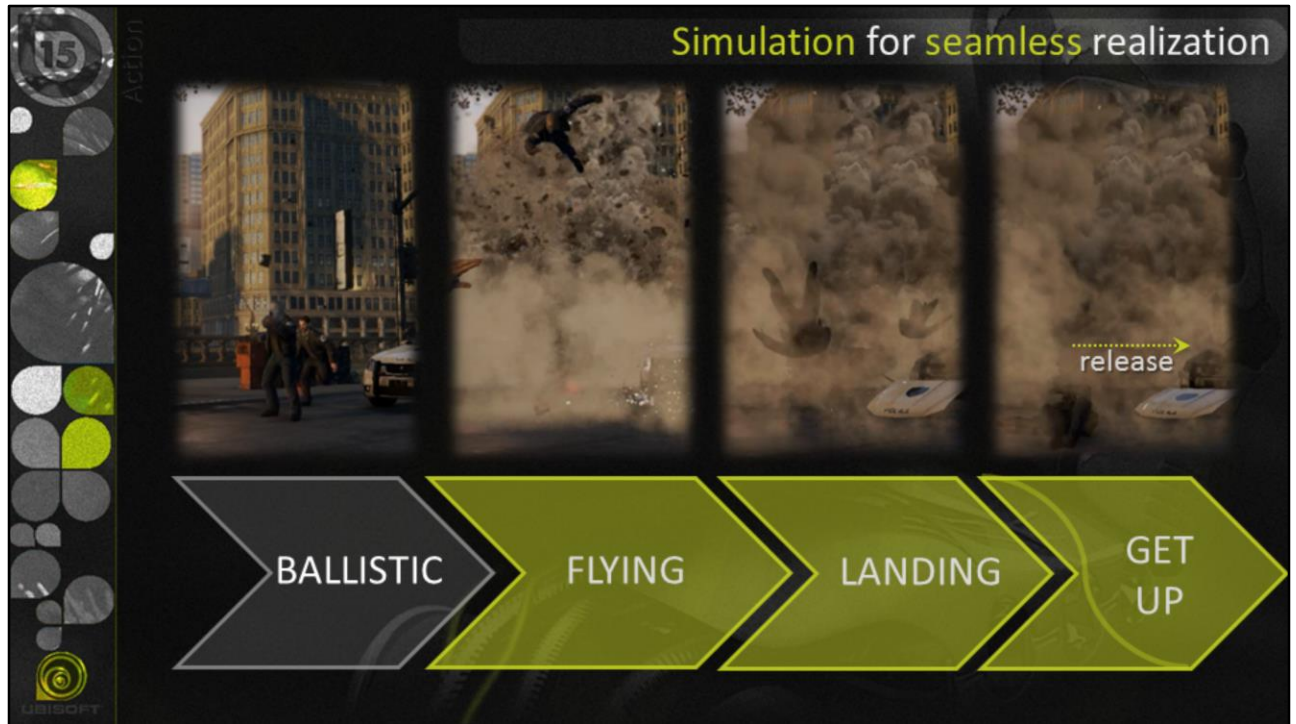


When receiving the impact stimulus, we compute right away the forces, the directions to start a ballistic simulation of the character propulsion in the air. We will **split this ballistic curve anticipation** with a certain resolution to **shape cast along it** to see where it is going to **hit something**. That way, we can predict in advance where the body will end up.



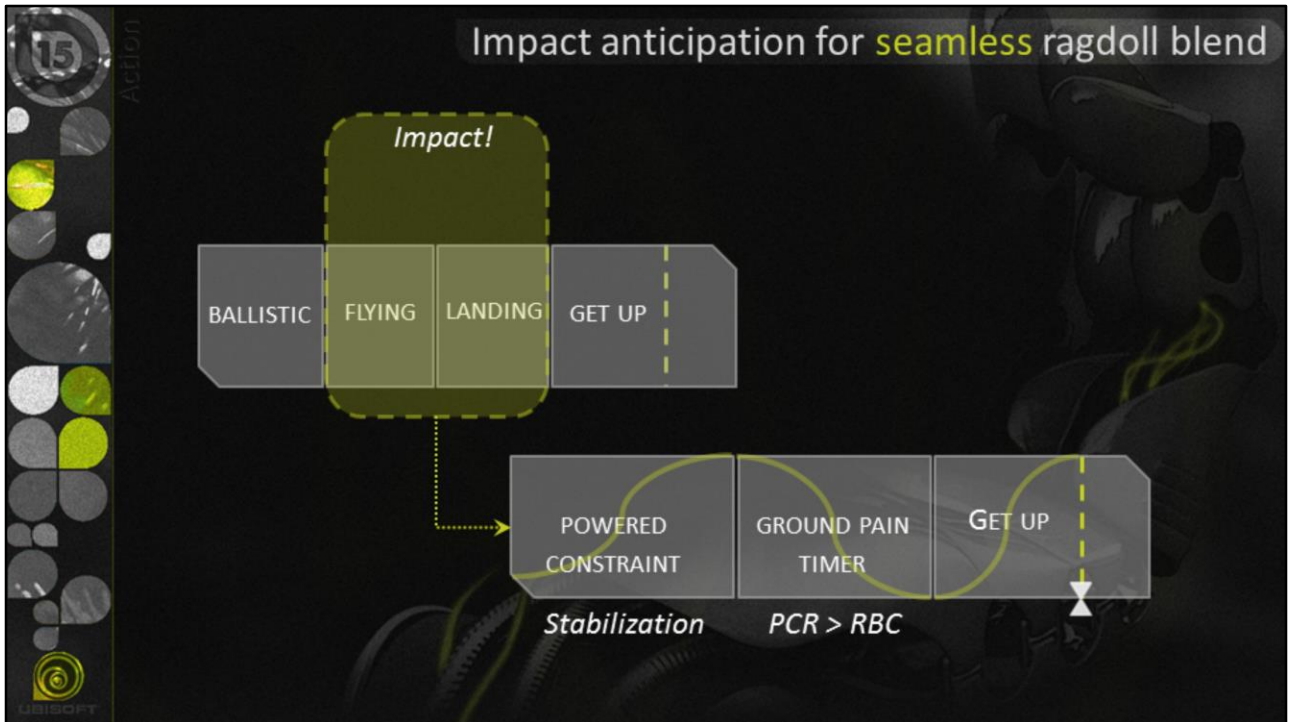
Once we have that, we can launch the crafted **parametric blended animation** and move it along the **computed ballistic curve**. The flying sub state will be responsible of pushing the **RTCP describing the flight** like the duration, the length, the apex, the angles, etc...

At the same time we launch the animation, we activate the **rigid body controller** of the ragdoll using the presets tweaked by the animator which are **ultra stiff for the flight pose**. We also continue to **cast a bit forward** along the line to detect dynamic objects because we need a bit of **anticipation** to blend properly between the animation and the ragdoll.



Being in the proper pose, we will be able to **blend** from the rigid body controllers to the get up animation fitting the **landing pose** and we will **progressively give back the control** to the player to move again

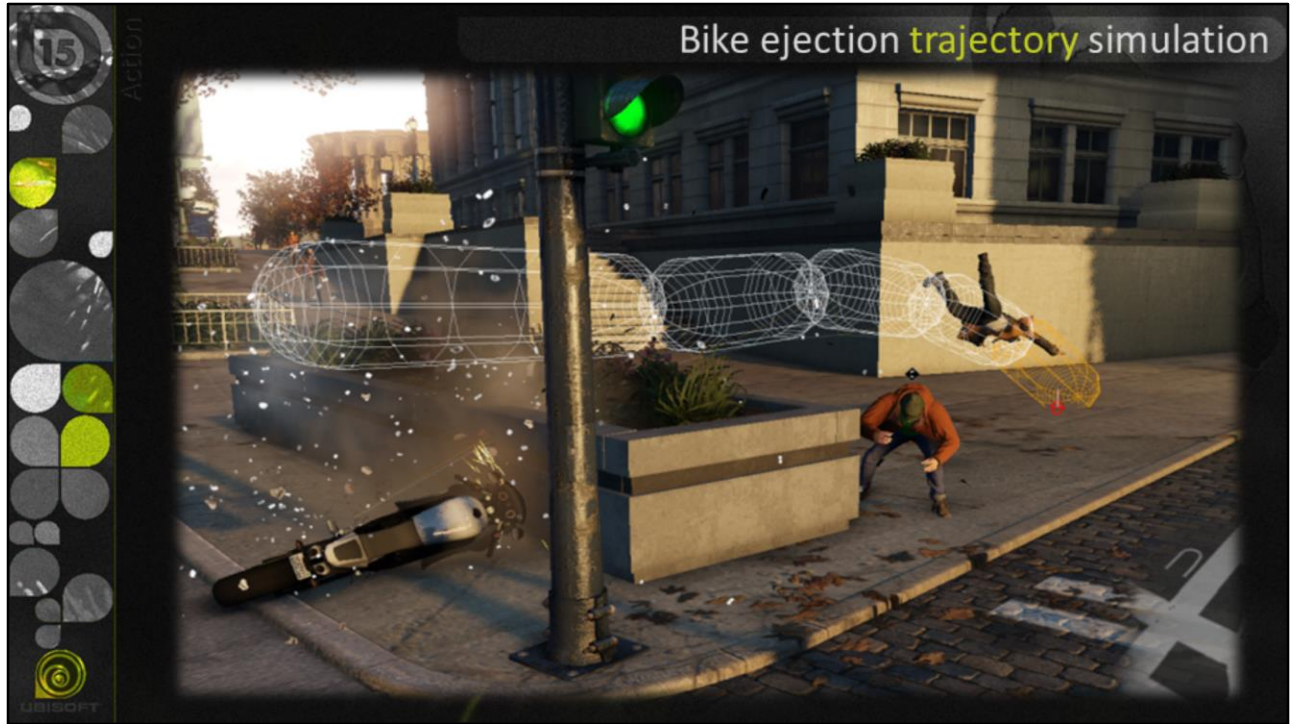




When anticipating an impact we will **blend the powered constraint ragdoll** based on the **animation ragdoll preset** and let it go to **collide with the environment** while trying to do a protecting pose.

Once stabilized, we will have to switch to ground pain to **blend from the powered constraint** to the **rigid body controllers** matching a **known get up pose**. This is done by checking the orientation of some **ragdoll bodies like the torso ones** since limbs will be replaced more easily by the Havok controllers.

Once in the proper pose, we will be able to **blend** from the rigid body controllers to the get up animation fitting the **analyzed pose** and we will **progressively give back the control** to the player to move again



Could be used for motorcycle crash ejection!

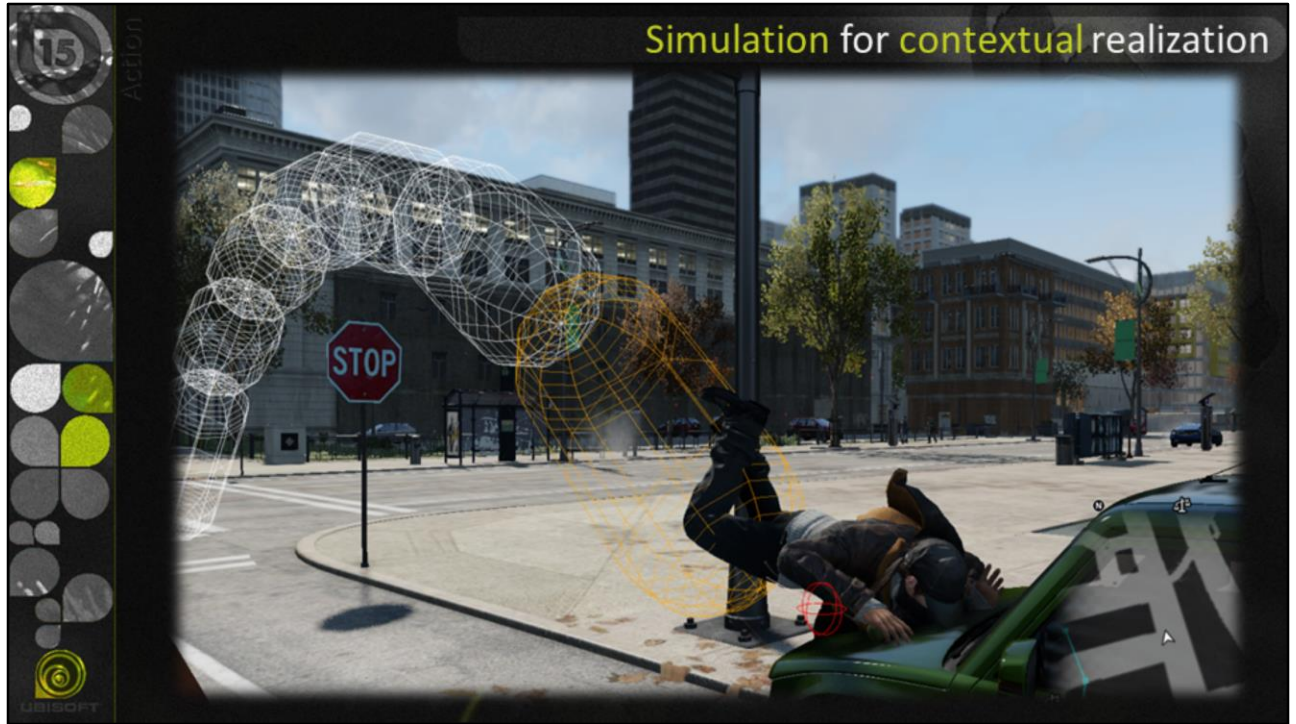


Or car bail out ejection!





Or, of course, grenade explosion propulsion!

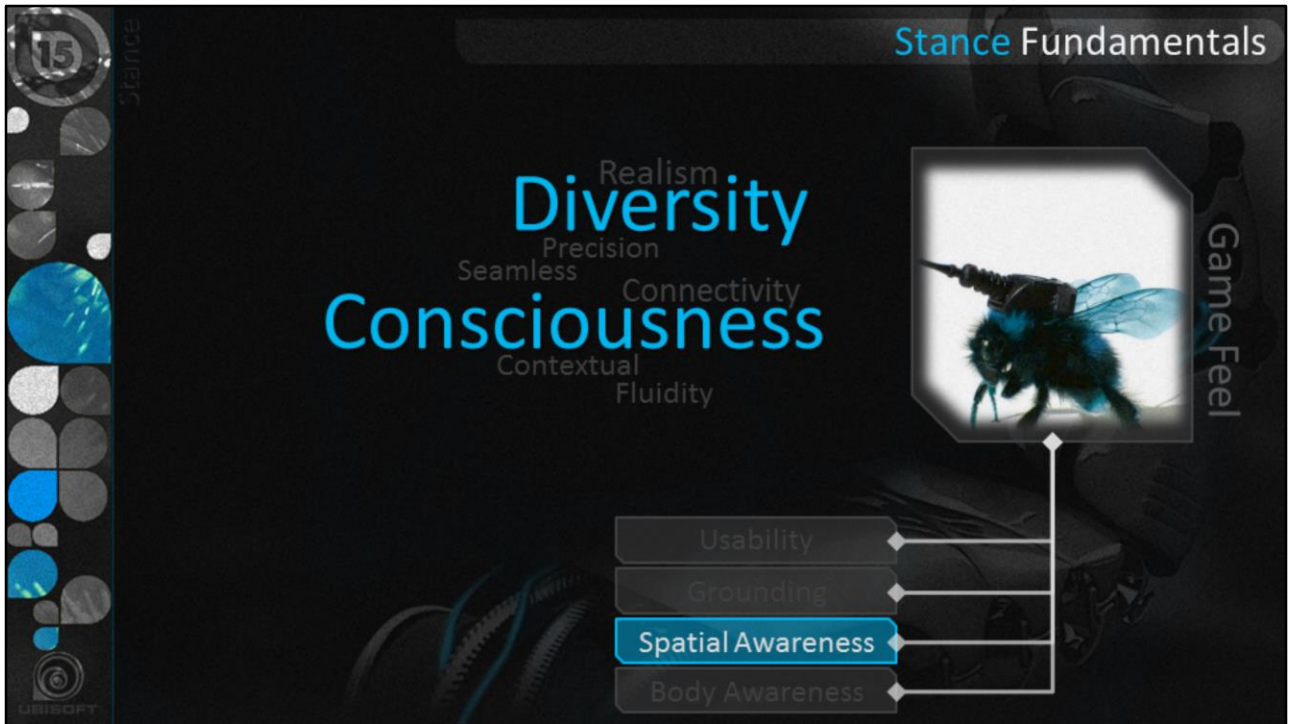


Which can be interrupted by **dynamic objects**! But in **most cases**, we will be **able to fully control the look** of it from the **beginning to the landing** without even activating the powered constraint ragdoll.



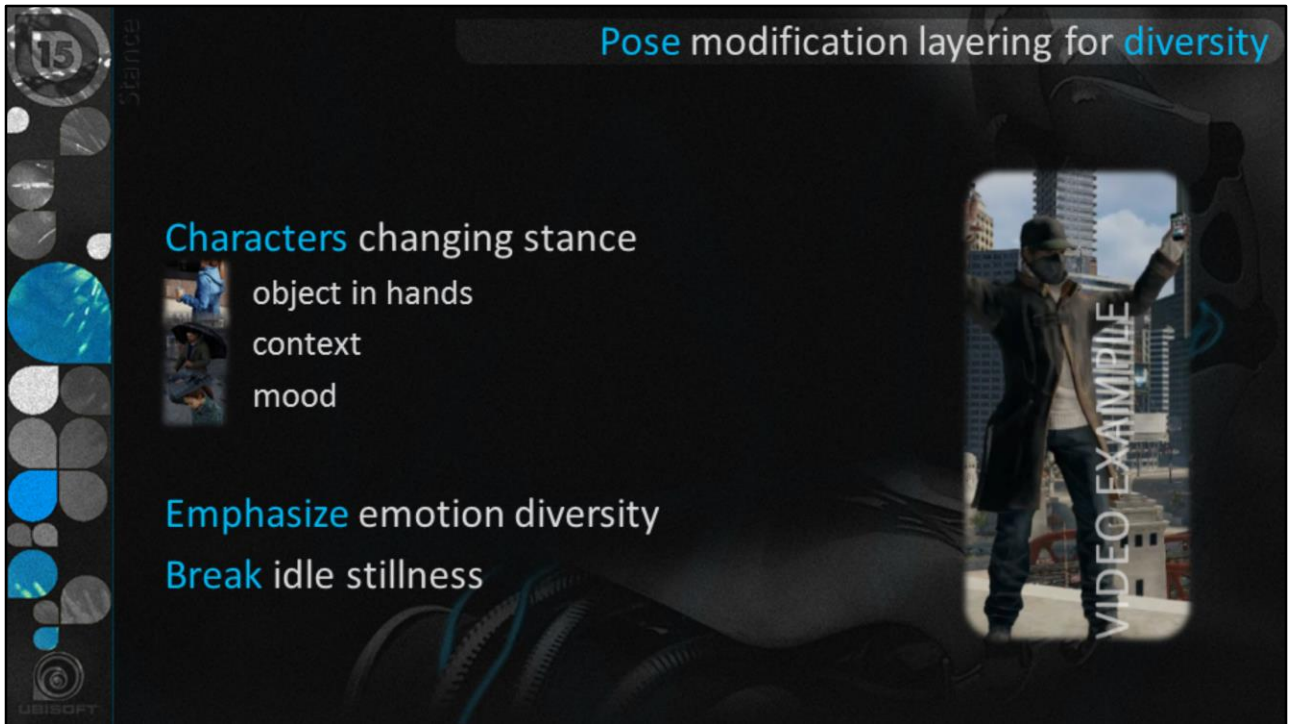
Here is a sequence of the game played with grenades while being invincible to showcase multiple angles and ballistic predictions and **multi-dimensional parametric blended animations**.





Let's follow with the **Stance** track which is supporting the **spatial awareness** aspect of the **game feel**.

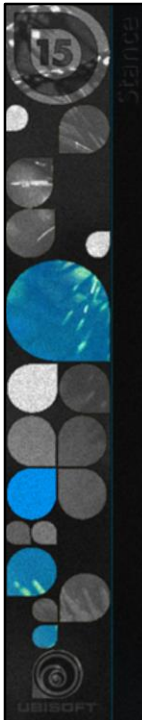
**Keywords** to keep in mind while building these states are **diversity** and **consciousness**!



We are talking about diversity because we want to feel the **various context** of characters.

This can be influenced by objects in **hand**, the **mood** or the **gameplay context**...

To **modify the pose** is really useful to **emphasize the emotion diversity** and to **break the idle stillness** of characters



## Pose modification layering for diversity

### Additive layering on full body

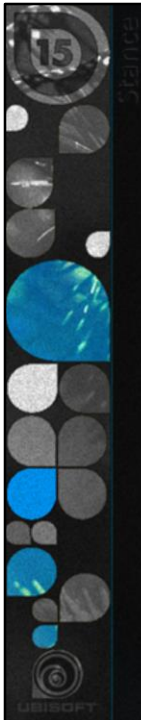
- different weights for each bone
- bone weights different for each speed
- feet IK resolution post layering
- ...not only useful for slopes!



Most of these pose modifications are done with **layered additive animations** playing on **top of the full body**

We expose **different weights for each bone** in the skeleton and we also expose the possibility to change the **bone weights at different speed** to change the way we are going to alter the full body animation. If you have already tried playing with additive animation **below the root**, you know that feet might go **through the ground** and this is why **Feet IK** is **not** only useful for **slopes** or to prevent feet from **sliding**...It is really useful to apply additive data on top of the legs and then resolve the **IK for the original locations** of the feet.





## Pose modification layering for diversity

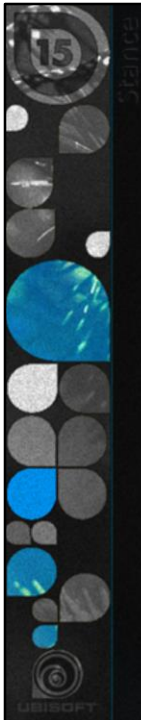
### Additive layering on full body

- different weights for each bone
- bone weights different for each speed
- feet IK resolution post layering
- ...not only useful for slopes!



Most of these pose modifications are done with **layered additive animations** playing on **top of the full body**

We expose **different weights for each bone** in the skeleton and we also expose the possibility to change the **bone weights at different speed** to change the way we are going to alter the full body animation. If you have already tried playing with additive animation **below the root**, you know that feet might go **through the ground** and this is why **Feet IK** is **not** only useful for **slopes** or to prevent feet from **sliding**...It is really useful to apply additive data on top of the legs and then resolve the **IK for the original locations** of the feet.



## Pose modification layering for diversity

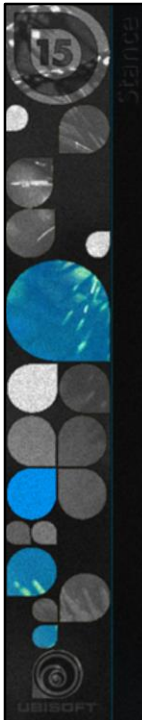
### Additive layering on full body

- different weights for each bone
- bone weights different for each speed
- feet IK resolution post layering
- ...not only useful for slopes!



Most of these pose modifications are done with **layered additive animations** playing on **top of the full body**

We expose **different weights for each bone** in the skeleton and we also expose the possibility to change the **bone weights at different speed** to change the way we are going to alter the full body animation. If you have already tried playing with additive animation **below the root**, you know that feet might go **through the ground** and this is why **Feet IK** is **not** only useful for **slopes** or to prevent feet from **sliding**...It is really useful to apply additive data on top of the legs and then resolve the **IK for the original locations** of the feet.



## Pose modification layering for diversity

### Additive layering on full body

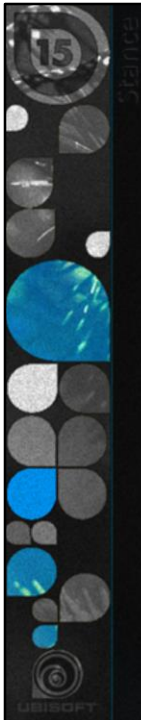
- different weights for each bone
- bone weights different for each speed
- feet IK resolution post layering
- ...not only useful for slopes!



Most of these pose modifications are done with **layered additive animations** playing on **top of the full body**

We expose **different weights for each bone** in the skeleton and we also expose the possibility to change the **bone weights at different speed** to change the way we are going to alter the full body animation. If you have already tried playing with additive animation **below the root**, you know that feet might go **through the ground** and this is why **Feet IK** is **not** only useful for **slopes** or to prevent feet from **sliding**...It is really useful to apply additive data on top of the legs and then resolve the **IK for the original locations** of the feet.





## Pose modification layering for diversity

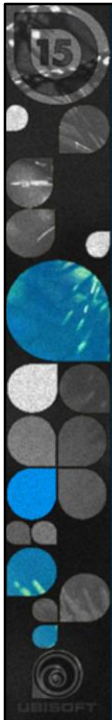
### Additive layering on full body

- different weights for each bone
- bone weights different for each speed
- feet IK resolution post layering
- ...not only useful for slopes!



Most of these pose modifications are done with **layered additive animations** playing on **top of the full body**

We expose **different weights for each bone** in the skeleton and we also expose the possibility to change the **bone weights at different speed** to change the way we are going to alter the full body animation. If you have already tried playing with additive animation **below the root**, you know that feet might go **through the ground** and this is why **Feet IK** is **not** only useful for **slopes** or to prevent feet from **sliding**...It is really useful to apply additive data on top of the legs and then resolve the **IK for the original locations** of the feet.



## Data driven look-at layering for consciousness

### Previous procedural approaches

- Fully in model space
- Fully in local space
- Blend of model & local space
- Cried...Decided to go **data driven** with custom spaces

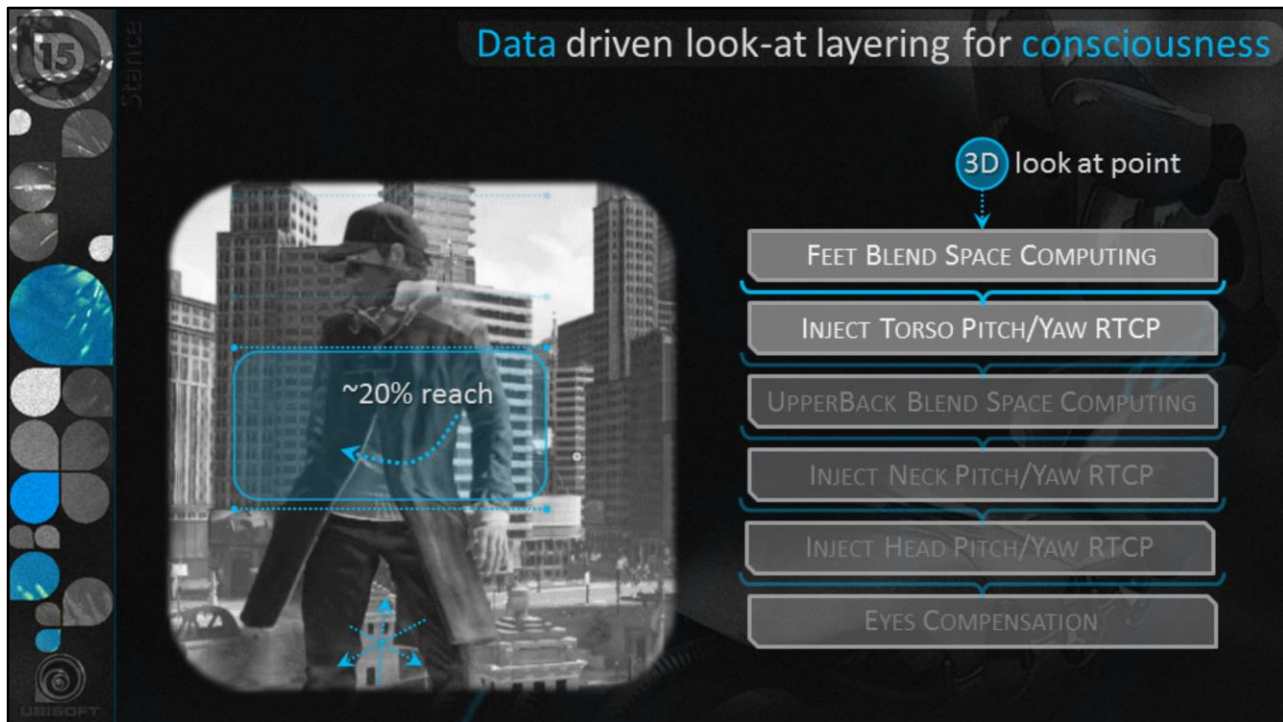


Enough for the pose modifications diversity, let's look at the technique we developed to **enhance the consciousness** of the character.

Initially we started by coding a procedural approach to do a look-at :

- Tried **model space** since it works well with **upright spine** characters but it was not looking good in **crouch stance**
- Tried **local space** to fix the **crouch stance** problems but it was still looking a bit **uncanny**
- Tried to **blend both ideas** but ended up with **a lot of use cases to tweak** in the end without any WOW visuals

Finally cried...decided to go fully **data driven** while using **custom made spaces** for the blending



The way we built the data for the look at is that we **split the body into multiple sections** and each of these sections are going to be dedicated to a portion of the look at. In our case, it is split into 3 parts, the **torso**, the **shoulders/neck** and the **head**. The evaluation of the **3D world anchor for the point of interest** will be done from the **bottom to the top**. Starting with the torso animation node to compute the **proportion of its rotation in pitch & yaw** that are going to be **RTCP internally injected**. And then the shoulders are going to be computed right after, continuing from where the torso stopped and converge a bit more based on the defined proportions and finally, the head, with the eyes, will complete the job at each frame. For each of these sections, the animators created a **multi-dimensional parametric blended animations** fitting within human constraints to move the proper bones for this section all around the required pitch & yaw.

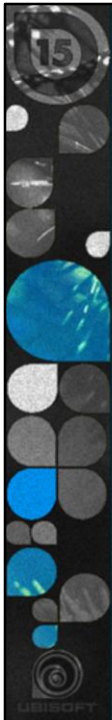




The way we built the data for the look at is that we **split the body into multiple sections** and each of these sections are going to be dedicated to a portion of the look at. In our case, it is split into 3 parts, the **torso**, the **shoulders/neck** and the **head**. The evaluation of the **3D world anchor for the point of interest** will be done from the **bottom to the top**. Starting with the torso animation node to compute the **proportion of its rotation in pitch & yaw** that are going to be **RTCP internally injected**. And then the shoulders are going to be computed right after, continuing from where the torso stopped and converge a bit more based on the defined proportions and finally, the head, with the eyes, will complete the job at each frame. For each of these sections, the animators created a **multi-dimensional parametric blended animations** fitting within human constraints to move the proper bones for this section all around the required pitch & yaw.



The way we built the data for the look at is that we **split the body into multiple sections** and each of these sections are going to be dedicated to a portion of the look at. In our case, it is split into 3 parts, the **torso**, the **shoulders/neck** and the **head**. The evaluation of the **3D world anchor for the point of interest** will be done from the **bottom to the top**. Starting with the torso animation node to compute the **proportion of its rotation in pitch & yaw** that are going to be **RTCP internally injected**. And then the shoulders are going to be computed right after, continuing from where the torso stopped and converge a bit more based on the defined proportions and finally, the head, with the eyes, will complete the job at each frame. For each of these sections, the animators created a **multi-dimensional parametric blended animations** fitting within human constraints to move the proper bones for this section all around the required pitch & yaw.



## FEET BLEND SPACE COMPUTING

```
// Get current referential
CQuaternion CRightFoot( sourceResult->m_displacementTransform.GetRotation()
* (modelCommonRaw->GetModelTransform( rightFootIdx, skeletonRaw, jointTransformArray ).GetRotation() * InvTSRightFoot) );

CQuaternion CLeftFoot( sourceResult->m_displacementTransform.GetRotation()
* (modelCommonRaw->GetModelTransform( leftFootIdx, skeletonRaw, jointTransformArray ).GetRotation() * InvTSLeftFoot) );

CQuaternion feetRot;
feetRot.QuaternionSlerp( CRightFoot, CLeftFoot, 0.5f );
feetRot.RemoveAllExceptAxis( CVector3( 0.f, 0.f, 1.f ) );
```

## UPPERBACK BLEND SPACE COMPUTING

```
// Get current referential
ndQuat upperBack( sourceResult->m_displacementTransform.GetRotation()
* (modelCommonRaw->GetModelTransform( upperBackIdx, skeletonRaw, jointTransformArray ).GetRotation() * InvTSUpperBack) );

m_lookatWorldTransform = ndPosQuatTransform( upperBack, neckPos );
```

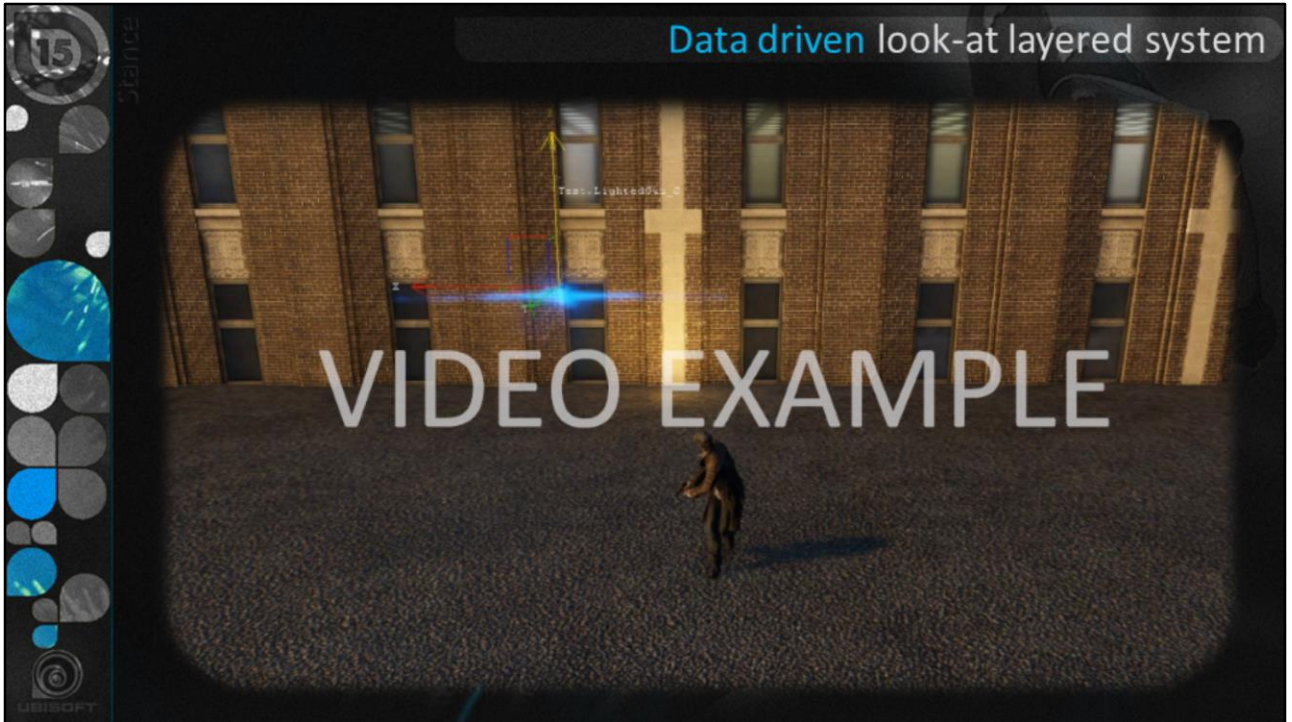
We said we built custom spaces for the blending of these stacked sections. In fact of all the things we tried, these **two** are worth to be used :

- **Feet averaged referential** for torso and shoulders parts
- **Upper back referential** for neck and head parts

That way it better supports the fact that **feet aren't necessary aligned with the model space**

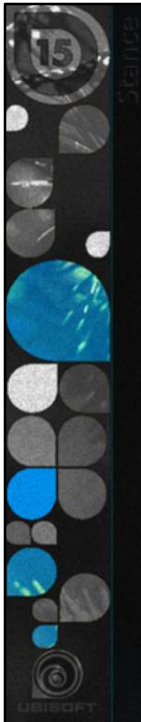
To extract them, you simply **multiply their transform** by the **inverse t-stance**





Now let's have a look at how it behaves with something moving around the character

As you can see, with the **same data set of parametric animations**, it fits properly on **multiple stances** and **multiple speeds**! You can even polish it more by tweaking the weights per situation if you want.



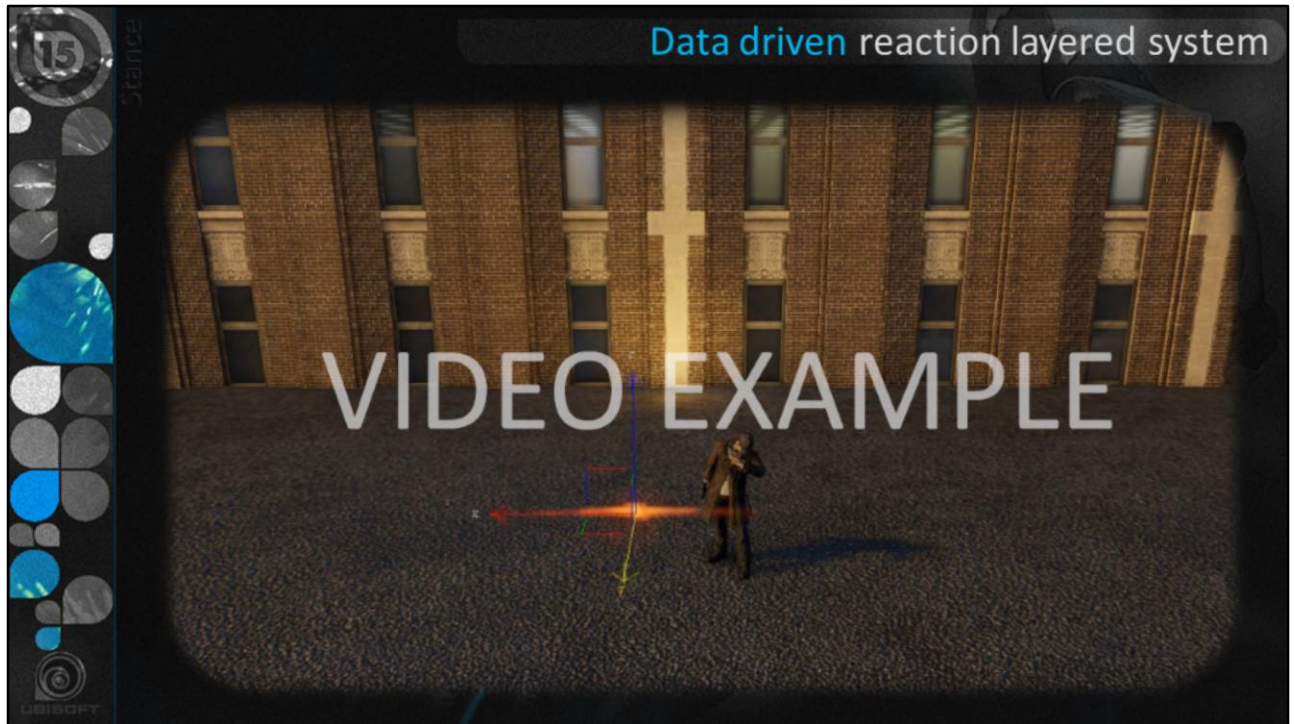
## Data driven look-at layering for consciousness

### Contextual data scalability

- distinction between glance & stare
- in car head look-at recycling
- anti-look-at for fire protection
- sky is the limit...  
(within animator & memory constraints)

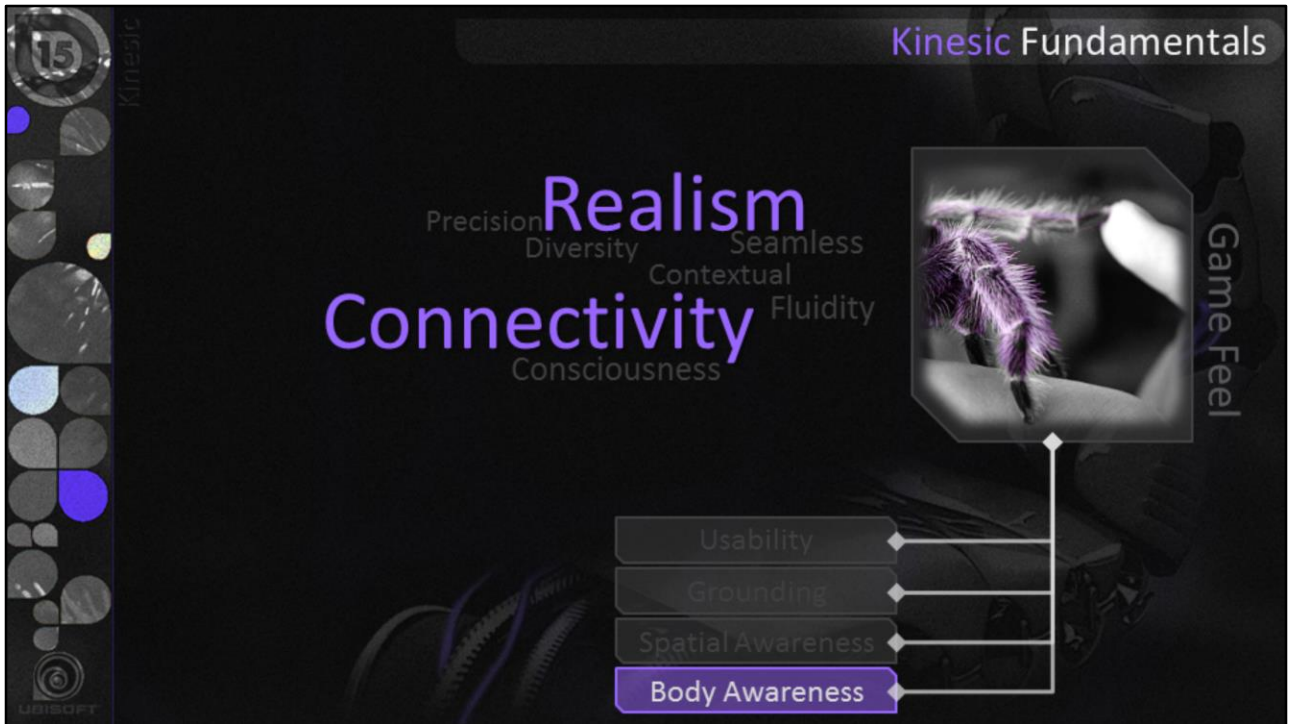


But this animation node approach can be used to other purposes. We can **recycle a portion of it** and use it in special stances like driving a car. The neck and head data will be used but we won't have the full proportion done by the torso since hands are mostly used for the steering wheel. We can also use for **the opposite of a look at like protecting** the character from something.



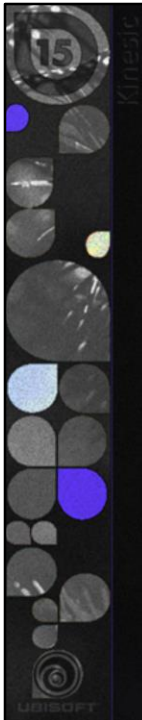
Here is a **glimpse** of the **same animation structure** with **different data** put in it protecting the character from some kind of fire





Let's follow with the **Kinesic** track which is supporting the **body awareness** aspect of the **game feel**.

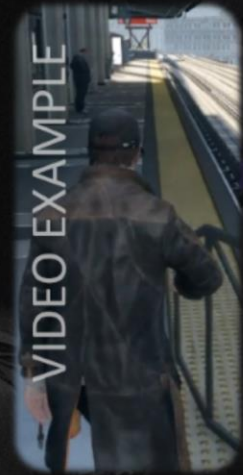
**Keywords** to keep in mind while building these states are **connectivity** and **realism**!



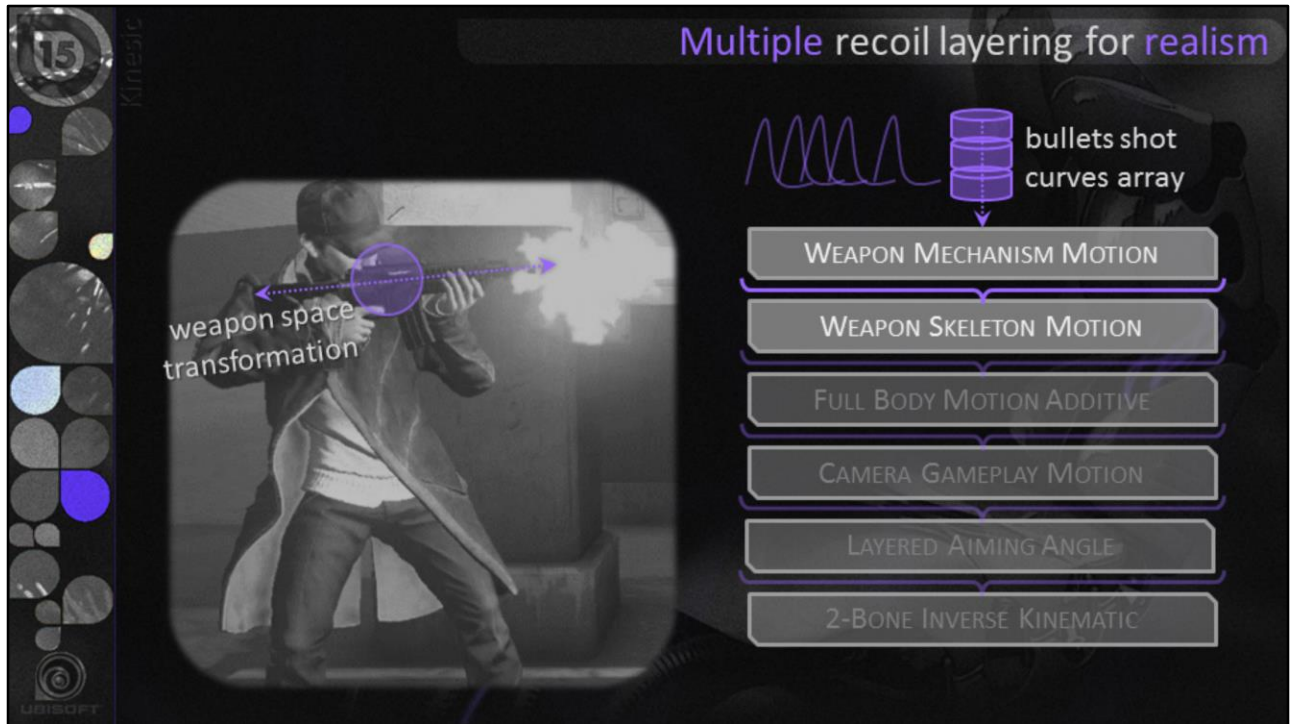
## Contacts with objects for connectivity

### Enhance the power of touching

- pushing doors
- grabbing grenades
- putting on the mask
- getting hands in pockets
- ...or even say hello to strangers!



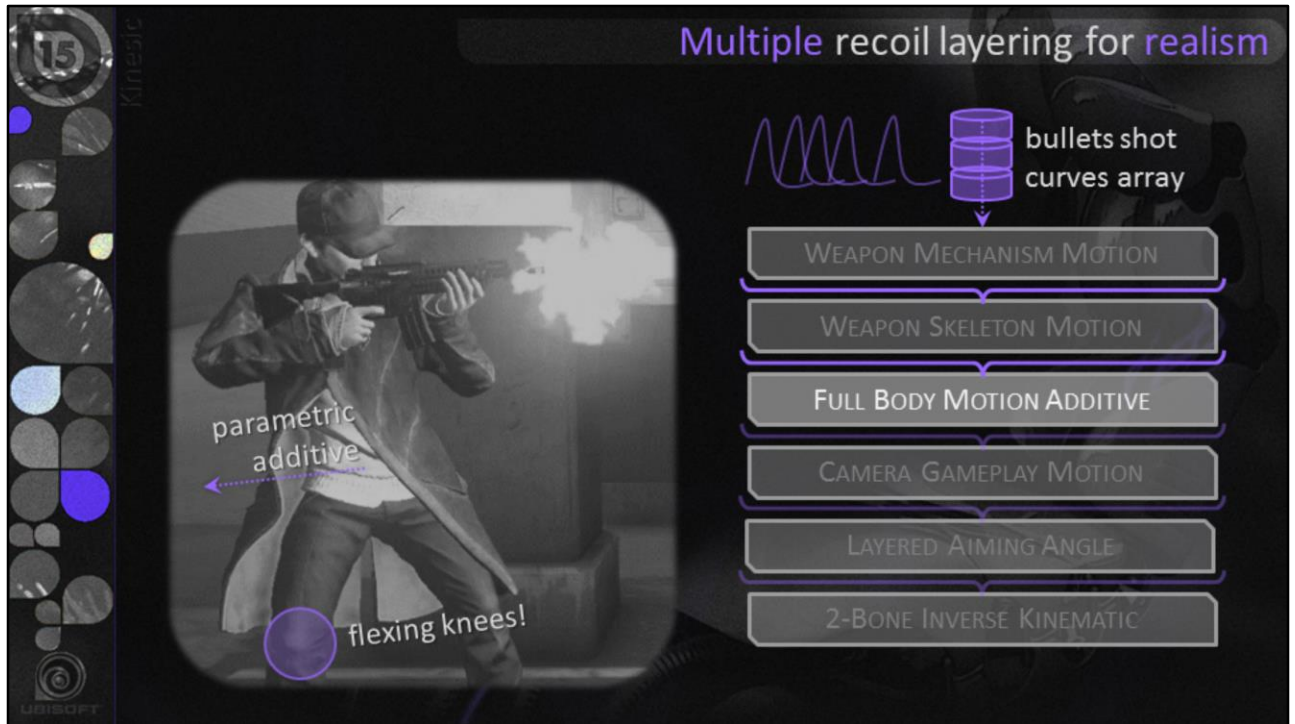
We never have too much contacts with the things on characters while in movement, we have to push this forward to **boost the immersion factor of the character and its body awareness**. We mainly did it for simple things like pushing doors, grabbing grenades, putting on the mask and putting hands in pockets that **enhanced the power of touching**. This is definitely an area where we could have done better on Watch\_Dogs though...



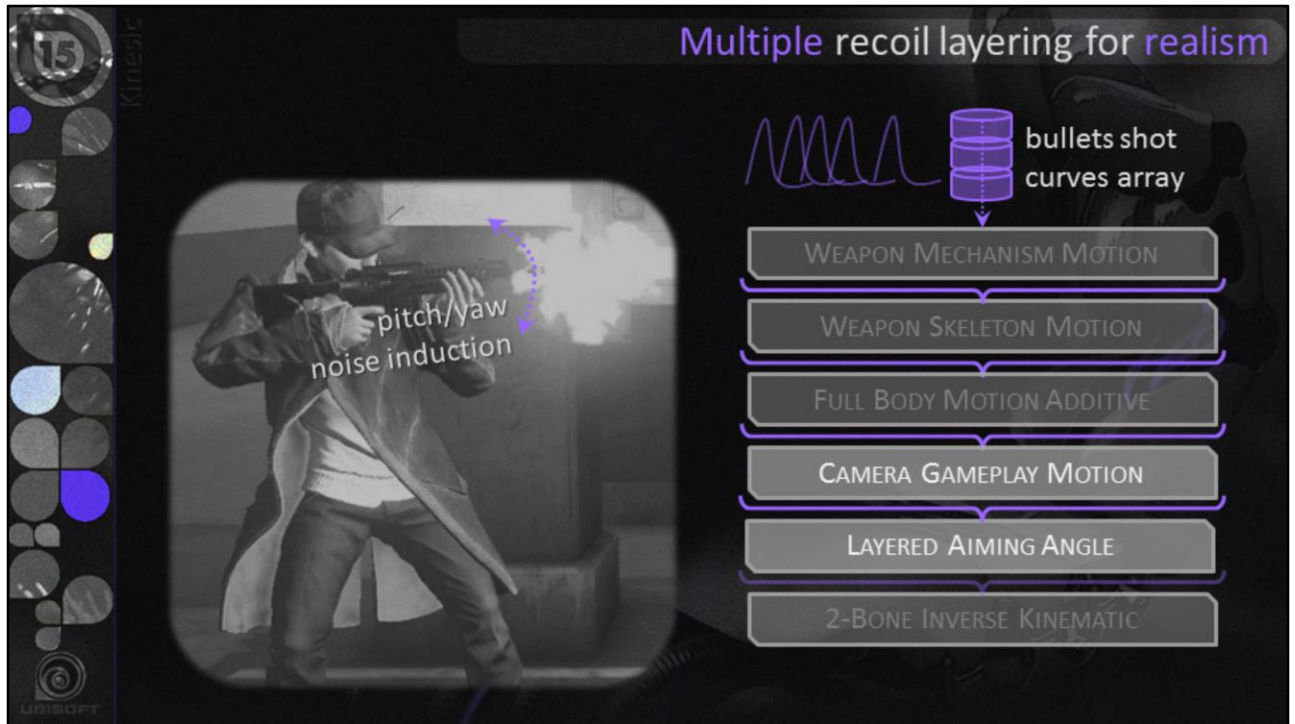
Talking about touching stuff with our hands, on the realism side of things, let's look at how firing a weapon was done animation wise, especially on the recoil.

- First, we exposed some curves in the editor for animators to **procedurally animate** the weapon when it fires
- Animators had control over **moving parts** of the weapon and the weapon itself in **weapon's model space**. That way they could **synchronize impacts** properly and because it was procedural, we were able on the code side to accumulate every curve playback into an array and stack them for a good feeling of accumulation.

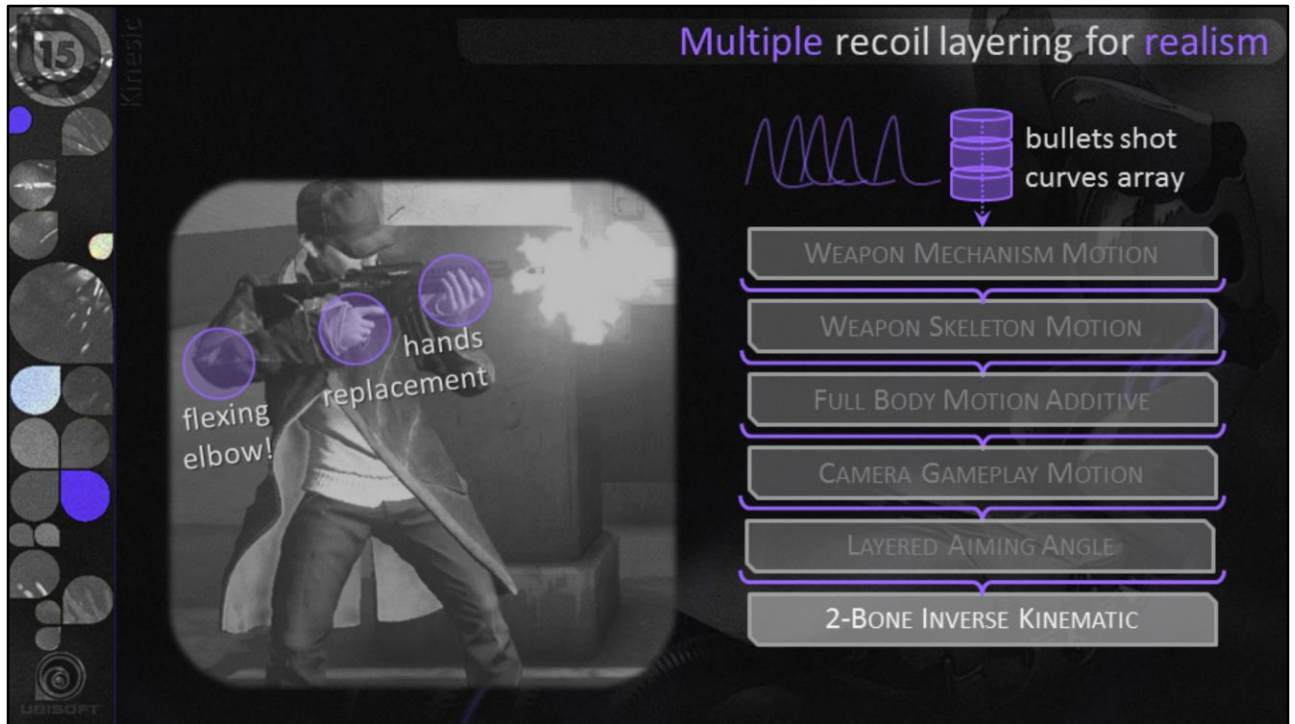




Then, animators also had **curves to drive directly some RTCPs** to influence the **full body parametric animation** they created when handling a weapon. That way, we could **feel the legs counterworking against the power** of the weapon. The angle of the **aiming** was purely done by an extra layer of **parametric animations** integrated with the full body pose in a multi-blending node to merge them properly



And then the camera was also adding noise to the experience to **simulate the out of control aspect of the weapon** firing too much. Designers were able to tweak with the same curve tool the noise per weapon how to go vertically and/or horizontally. And then the aiming animated pose would adjust based on the camera direction.

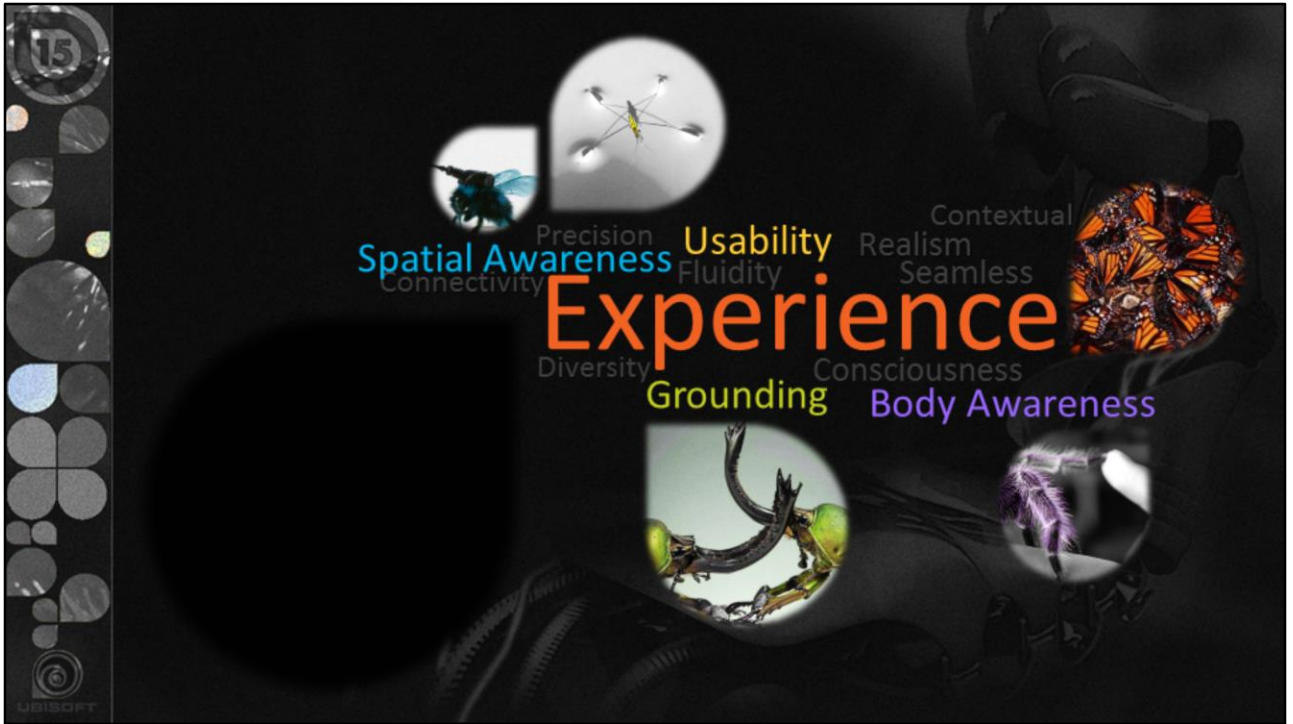


Finally, in the post-physics animation update pass, we would complete the job by computing the **2-bone IK** of both hands to fit the newly placed weapon while taking into account the body modifications done by the animations

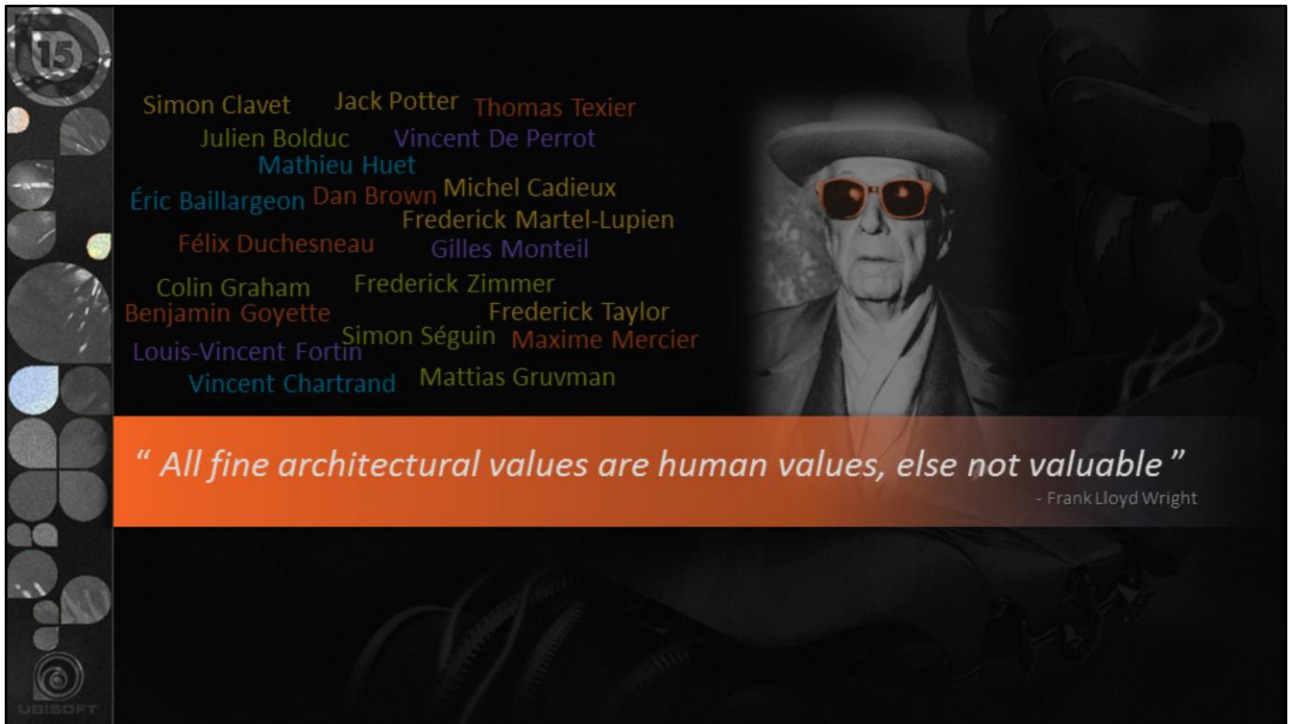




Here is the side view of the character using a rifle



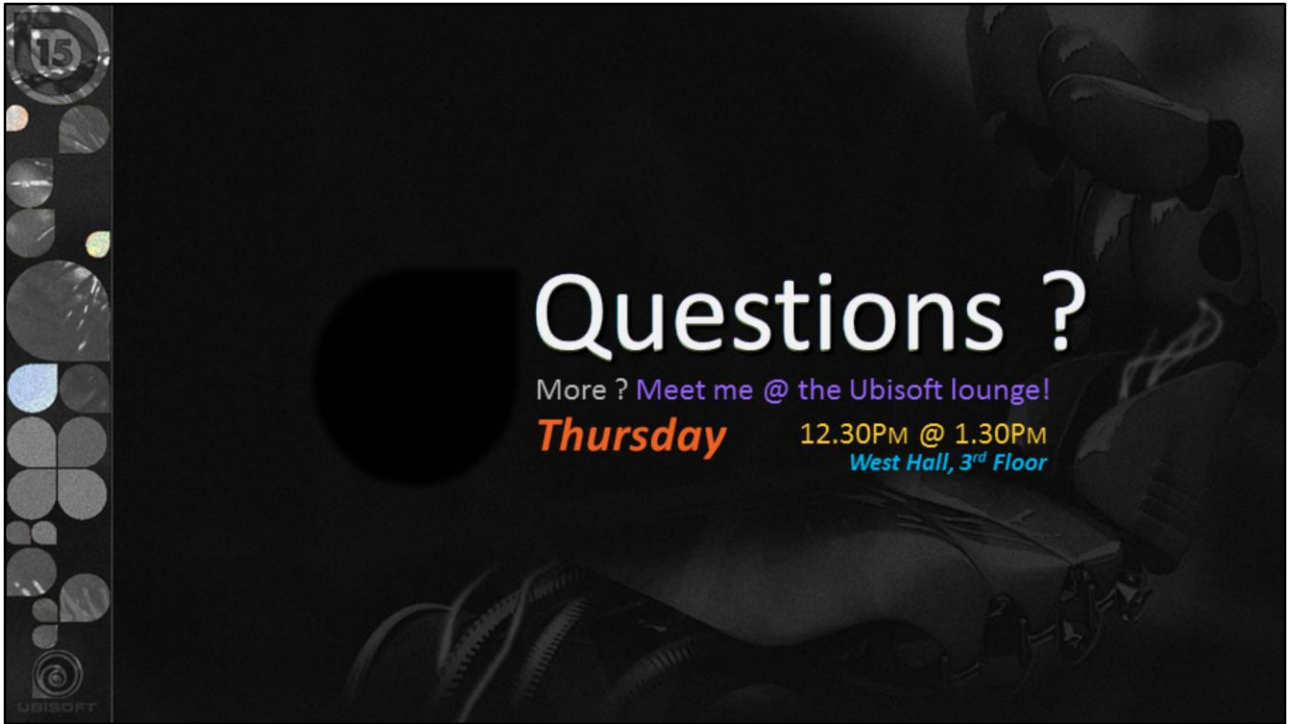
This concludes the talk...So let's not forget about the player's experience when building player mechanics. Always keep in mind usability, grounding, spatial awareness and body awareness. I hope you learned interesting tips and I hope you will steal them because I stole from a lot of people over the last years...



Which leads me to another quote of Frank...

To thank a lot of people I crossed or worked with to achieve this work.





For deeper questions, feel free to come and see me tomorrow at the Ubi Lounge!